The Book of C

Release 2018.01

Joel Sommers

Sep 16, 2018

CONTENTS

This book is also available in PDF form.

Fig. 1: New Harbor, Maine, USA. Photo by J. Sommers.

Preface

Get ready to learn one of the most influential programming languages ever developed. If you know some Java, you'll find C's syntax familiar (Java's syntax is based on C) and many of the same control structures. That familiarity can be deceptive, though. C is rather unforgiving and will allow you (nay, give you the weapon) to shoot yourself in the foot[1]. On the other hand, you'll develop a much better understanding of computer systems as your knowledge of C grows. Have fun, and good luck!

This book is loosely based on the "Essential C" document written by Nick Parlante at Stanford University. The original document is available at http://cslibrary.stanford.edu/101/. The Essential C document was last updated in 2003, which is one reason why this document exists. Programming languages and compilers change, and I wanted to take the good work that Nick had done and make several updates to modernize the text. The notice reproduced below is copied from Essential C:

> **Stanford CS Education Library**. This is document #101, Essential C, in the Stanford CS Education Library. This and other educational materials are available for free at http://cslibrary.stanford.edu/. This article is free to be used, reproduced, excerpted, retransmitted, or sold so long as this notice is clearly reproduced at its beginning.

[1] http://www.toodarkpark.org/computers/humor/shoot-self-in-foot.html

CHAPTER

ONE

INTRODUCTION: THE C LANGUAGE

C is a professional programmer's language. It is a *systems implementation language*, and was originally designed by Dennis Ritchie at Bell Labs in the 1960s. Prior to the development of C, most operating systems were developed in assembly language[1]! The C language (and its predecessor, the B language) were designed to provide some level of portability of the operating system source code (i.e., it could be recompiled for a new processor architecture), while still allowing a programmer to manipulate low-level resources such as memory and I/O devices. If you are interested in some the history behind the development of UNIX, see [Evolution] by Dennis Ritchie.

C was designed to get in one's way as little as possible. It is a high-level programming language (i.e., not assembly), but is quite minimal in design and scope. Its minimalism can make it feel difficult to work with. Its syntax is somewhat terse, and its runtime environment does not contain the sort of "guardrails" available in other programming languages such as Python or Java. There are no classes, objects, or methods, only functions. C's type system and error checks exist only at compile-time. The compiled code runs with no safety checks for bad type casts, bad array indexes, or bad memory accesses. There is no garbage collector to manage memory as in Java and Python. Instead, the programmer must manage heap memory manually. All this makes C fast but fragile.

Some languages, like Python, are forgiving. The programmer needs only a basic sense of how things work. Errors in the code are flagged by the compile-time or run-time system, and a programmer can muddle through and eventually fix things up to work correctly. The C language is not like that. The C programming model assumes that the programmer knows exactly what he or she wants to do, and how to use the language constructs to achieve that goal. The language lets the expert programmer express what they want in the minimum time by staying out of their way.

As a result, C can be hard to work with at first. A feature can work fine in one context, but crash in another. The programmer needs to understand how the features work and use them correctly. On the other hand, the number of features is pretty small, and the simplicity of C has a certain beauty to it. What seem like limitations at first can feel liberating as you gain more experience with the language.

As you start to learn and use C, a good piece of advice is to just *be careful*. Don't write code that you don't understand. Debugging can be challenging in C, and the language is unforgiving. Create a mental (or real) picture of how your program is using memory. That's good advice for writing code in any language, but in C it is critical.

C's popularity

Although introduced over 40 years ago, C is one of the most popular programming languages in use today[2].Moreover, C's syntax has highly influenced the design of other programming languages (Java

[1] Take a moment and consider how difficult it must have been to develop a new OS!

syntax is largely based on C). All modern operating systems today are written largely in C (or a combination of C and C++) due to the fact that low-level resources (memory and I/O devices) can be directly manipulated with a minimum of assembly.

This book is intended to be a short, though mostly complete introduction to the C programming language. A (generally) C99-capable compiler assumed since the book introduces various features from the C99 revision of the language [C99]. For an in-depth treatment of the language and language features, there are two other books to recommend. The mostly widely used C book is one simply referred to as "K&R", written by the designers of the language, Brian Kernighan and Dennis Ritchie [KR]. It is an excellent (though somewhat dated) reference to the language. An excellent, if lengthy, introduction to C can be found in Stephen Prata's *C Primer Plus* [CPP]. It contains a more modern treatment of C than K&R, with lots of detail and exercises.

References

[2] See http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html for one ranking of programming language popularity.

CHAPTER

TWO

GETTING YOUR FEET WET IN THE C

In this first chapter, we'll get wet in the C by way of a tutorial that examines a variety of topics that are discussed in depth in later chapters. Where appropriate, pointers to later sections are given if you want to read a bit more as you work through the following examples.

2.1 Hello, somebody

We first start out by examining a simple "hello, world"-style program, but with a twist to make it somewhat more interesting. The following program asks a user for his or her name, then prints `Hello, <name>!`. Here is the code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    printf("Enter your name: ");
    char name[32];
    fgets(name, 32, stdin);
    name[strlen(name)-1] = '\0';
    printf("Hello, %s!\n", name);
    return EXIT_SUCCESS;
}
```

Since this may be the first C program you've seen, we'll walk through it line by line:

1. The first line (`#include <stdio.h>`) tells the C compiler that we intend to use functions that are in the *C standard library* and declared in a *header file* called `stdio.h`. This line is (somewhat) analogous to saying `import` in Python or Java. The file `stdio.h` contains declarations for input/output functions in C, such as `printf` and `fgets`, which we will use to print text to the console and get input from a user, respectively. Including those declarations in our program lets the compiler know the type signatures of the functions we want to use (i.e., the data types of the function parameters, and the return type). See *preprocessing* for more discussion of `#include` files.
2. The second line includes another source of standard function declarations and definitions (`#include <stdlib.h>`). We include this file specifically for the `EXIT_SUCCESS` value returned on the last line of the `main` function (as discussed below).
3. The third line is another `#include` line to tell the C compiler that we want to use some string-related functions. In particular, we use `#include <string.h>` because we want to use a function called `strlen`, which is used to compute the length of a string.
4. Whitespace is generally ignored in C, just as it is ignored in Java.

5. Line 5 starts a function called `main`, which is the function in which every C program **must** start. If you don't define a `main` function, the compiler will complain loudly. Notice that the return value of `main` is of type `int`. In UNIX-based systems (and C), it is common for functions to return an integer value indicating whether the function "succeeded" or not. Weirdly enough 0 usually means "success" and some non-zero value (sometimes 1, or -1, or some other value) means "failure". The symbol `EXIT_SUCCESS` is defined (inside `<stdlib.h>`) to be 0; there is also a symbol `EXIT_FAILURE` that is defined to be 1. If you look at line 11, you'll see that we unconditionally return `EXIT_SUCCESS` to indicate that the program successfully completes. (The `main` function can also take two parameters; see *the main function* for more about `main`.)

6. We print a prompt for a user using the `printf` function. There shouldn't be anything particularly surprising about this function call: as in Java, the string we print out is delimited with double quotes ("). In C, all strings **must** be delimited with double quotes, unlike Python which has three different ways of enclosing text (single, double, and triple quotes). (Note that, like Java, single characters must be enclosed in single quotes in C.) You can also see on line 5 that the `printf` statement is terminated with a semicolon. As with Java, all statements in C must end with a semicolon. The `printf` function can take any number of parameters, and a "format" string can define how to display different data types. See *stdio reference* for more.

7. On line 7, we declare a variable called `name`, which is an array of length 32 of `char`. A string in C is represented as an array of characters, where character after the last valid character of the string contains the special character `'\0'`. This character is referred to as the "null" character and is used to delimit strings. The string termination character must fit within the array we've declared here, so the array we've declared can hold a maximum of 31 characters plus the null termination character. (Note: don't confuse the null character with the `NULL` pointer, which we'll encounter later — they're totally different.) Unlike any other language you have probably encountered so far, the *representation* of a string in C is completely transparent: there is no "information hiding" or abstraction here — it is all visible. Thus, unlike strings in Python and Java which are *immutable*, strings in C are completely mutable. Changing a string in C boils down to modifying individual characters of the string. You just need to be sure that the string is terminated with the special `'\0'` (null) character. See *character literals* for more information on character literals and the null character.

   One thing that may not be obvious on line 7 is that this statement only declares the variable `name`. The C compiler and runtime environment do not do any automatic initialization of the string. So at the point of declaration, the programmer cannot assume that there is anything known about the contents of `name` — it is currently what ever random characters that may have resided in the memory location occupied by `name` prior to its being brought to life.

8. The next statement calls the `fgets` C library function to get input from the keyboard. There are three parameters we pass: the string variable (i.e., the array of characters) we created on line 7 (`name`), the maximum number of characters that `name` can hold (32), and the "input stream" that we wish to read from. The variable name `stdin` is predefined in the `<stdio.h>` header file, so we can just use it directly as the third parameter. `stdin` means "standard input", and (usually) refers to input that can be collected from the keyboard. As you'll see later on in this book, you can also use `fgets` to read from files that you open. See *stdio reference* for more.

   One note about the way `fgets` works: it will read up to *one less* than the number of characters you specify as the second parameter. It must reserve space for the final null termination character (since all C strings must end with `'\0'`). If a user types more than 31 characters, we'll just get the first 31.

And one more note about C strings: they have a fixed maximum size. Unlike Python or Java strings, C strings cannot automatically grow or shrink as needed — this is left entirely to the programmer to deal with. As a result, a programmer must assume some reasonable maximum length for string input. This issue is discussed further in *dynamic strings*.

9. One more important thing about the way `fgets` works is that if a user types a name followed by return, the character emitted when the return key is pressed is also included in the string. This character happens to be the lowly newline (`'\n'`). The task of line 9 is to remove this character so that we only have a person's name stored in `name`, not including the newline character. To get rid of the newline, we simply overwrite the last character of the string with a `'\0'` (remember: in C, strings are totally mutable!)

   This line is not foolproof. If the user types in more than 31 characters, the newline won't be included in `name`. As a result, line 9 will squash a "legitimate" character of the name. It is left as an exercise to fix this potential problem.

10. On line 10 the "Hello" statement is printed using the `printf` function, which is built into the C standard library of functions. Two things to note about this line of code: the first parameter defines a *format template*, including the special character sequence `%s` that tells `printf` that a C string should be substituted at that point in the template. The string to be substituted is the next parameter supplied to `printf`, which is just the variable `name`. Secondly, we have included an explicit newline character in the format string. `printf` never automatically emits a newline for you: if you want one, you need to remember to include `'\n'`. We did not include one in the first `printf` statement (line 5) because we wanted the user to be able to type the name on the same line as the prompt. Note that two other common `printf` formatting placeholders are `%d` for printing a decimal integer, and `%f` for printing a floating point number.

11. See the description of line 5, above.

2.2 Hello, `clang`

Now that we've gone through the `hello` program in some gory detail, let's compile and run it. Just like Java, C requires an explicit compilation step to produce an executable program. Unlike Java, however, the program produced by the compiler does not contain "byte code", it contains *real* executable instructions for the processor on which you're running the compiler. So, if you happen to be compiling on a 64-bit Linux system using an Intel-based processor, the program produced by the C compiler will contain raw Intel x86_64 instructions[1].

All the examples in this book will use a compiler called **`clang`**, and all examples will use the command line (in particular, the **`bash`** shell). **`clang`** is available on most recent Linux systems and on MacOS X[2]. If **`clang`** is unavailable, you can also use the **`gcc`** compiler. The reason we favor **`clang`** is that the error messages produced by **`clang`** are far, far superior to the cryptic nonsense spewed by **`gcc`**.

The basic recipe for using **`clang`** is:

```
clang -g -Wall -std=c99 -o <executablename> <sourcefiles>
```

Notice that there are a few command-line flags/options given:

- `-g` tells the compiler to include debugging symbols in the compiled program. This is a good thing to do because it will enable you to use a symbolic debugger like **`gdb`**, if necessary.

[1] The file containing the instructions is in a format known as ELF: http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[2] Don't look for any examples related to Windows or Visual C in this book: they don't exist.

- `-Wall` tells the compiler to turn on all warnings. If **clang** detects something odd or suspicious about your code, it will say so. Turning on this flag is a ridiculously good idea. If you enjoy the feeling of someone yelling at you, you can even turn on the `-pedantic` flag.
- `-std=c99` tells the compiler to turn on "C99" features, or language features that were introduced in a 1999 revision of the C programming language standard. We will use various C99 constructs in this book, so you should always turn on this flag[3].
- `-o <executablename>` tells the compiler how to name the file that is produced as an executable program. You should replace `<executablename>` with something more meaningful (as shown in the example below).
- `<sourcefiles>` comes last, and can be one or more `.c` files containing C source code.

a.out is the default executable file name produced by clang

If you do not supply an output executable name using the `-o` flag, **clang** will create a file named `a.out`. Which is very, very weird, right? There is, of course, a history behind this file name[4].

For example, with the "hello, someone" example above, we might compile and run the program as follows:

```
$ clang -g -Wall -std=c99 -o hello hello.c
$ ./hello
Enter your name: fred
Hello, fred!
```

When invoked like this, **clang** will perform all three basic phases of compilation (see *compilation phases* for details): preprocessing, compiling, and linking. The final result is a binary executable program that can be executed on a particular processor architecture (e.g., Intel x86 or x86_64).

2.2.1 When `clang` goes "bang!"

Of course, not all our wonderful source code will compile and run correctly on the first go. Let's modify our "hello, somebody" program to introduce a bug and see what happens. Specifically, we will modify two lines: on line 6 we will remove the semicolon at the end of the line, and on line 9, we will remove the second argument to the `printf` function call (the line should then read: `printf("Hello, %s!\n");`). Here's what **clang**'s output looks like:

```
hello.c:6:18: error: expected ';' at end of declaration
    char name[32]
                 ^
                 ;
hello.c:9:21: warning: more '%' conversions than data arguments [-Wformat]
    printf("Hello, %s!\n");
                    ~^
1 warning and 1 error generated.
```

Clang helpfully tells us that we're missing the semicolon (the 6:18 means the sixth line and 18th character on that line), and that there was an unequal number of arguments to `printf` and `%`-placeholders in the format string[5].

[3] By default, recent versions of **clang** operate in a "modern" mode (C99 or C11). To be safe, however, you should specify the C standard to which the code is written.

[4] http://en.wikipedia.org/wiki/A.out

[5] If the terminal in which you invoke **clang** is capable, it will even color-highlight the output to help draw your attention to various errors and warnings.

2.2.2 `clang` versus `gcc`

If you use **`gcc`** instead of **`clang`** (perhaps because **`clang`** is not available for some reason), the command-line options are *exactly* the same. Nothing needs to change there. The key difference you will notice between **`gcc`** and **`clang`** is in the error and warning messages. If you thought the error messages above weren't very good, ponder the following output of **`gcc`** for the same example:

```
hello.c: In function âĂŸmainâĂŹ:
hello.c:7:5: error: expected âĂŸ=âĂŹ, âĂŸ,âĂŹ, âĂŸ;âĂŹ, âĂŸasmâĂŹ or âĂŸ__attribute__âĂŹ before␣
↪âĂŸfgetsâĂŹ
     fgets(name, 32, stdin);
     ^
hello.c:7:11: error: âĂŸnameâĂŹ undeclared (first use in this function)
     fgets(name, 32, stdin);
           ^
hello.c:7:11: note: each undeclared identifier is reported only once for each function it appears␣
↪in
hello.c:9:5: warning: format âĂŸ%sâĂŹ expects a matching âĂŸchar *âĂŹ argument [-Wformat=]
     printf("Hello, %s!\n");
 ^
```

Ugh. Not only are there *more* errors reported than actually exist, the output is simply more cluttered and confusing. Advice: use **`clang`** instead of **`gcc`** if at all possible.

Exercises

1. A common error is to fail to include a null-termination character in a C string. Modify the code example above to remove the `'\0'`. What does `printf` do? What do you think is happening? (Note: you actually have to work a bit to remove any null-termination character, since `fgets` will automatically null-terminate a string. You can either overwrite the null-termination character added by `fgets` with some other character, or devise some other method.)
2. Revise the program above to change the size of the array to something small (e.g., 4 characters), then recompile and re-run it. What happens when you input a string that's longer than 4 characters? Exactly 4 characters?
3. Line 8 in our example program above can squash a non-newline character if the user types more than 31 characters for a name. Fix the program so that the last character of the string is *only* overwritten if it is a newline (`\n`). Note that, as with Java and Python, characters can be compared using == (double-equals), and that `if` statements work very similarly in C as they do in Java.
4. Revisit your fabulous time in COSC 101 by writing a program that asks for a dog's name, its age in human years, then prints its age in dog years (human years multiplied by 7). A couple hints to help accomplish this:
   - You can use `fgets` to collect each input, but note that you'll need to convert the dog age to an integer (you can just do the computations as integers). You can use the `atoi` function to convert a string to an integer; `atoi` takes a C string as a parameter and returns an `int`. You'll need to `#include <stdlib.h>` to use the `atoi` function.
   - Arithmetic operators in C are virtual identical to those available in Java (and Python). There is no ** operator as in Python to do exponentiation, but you certainly shouldn't need that to compute the dog's age.
   - Try to make the output look nicely formatted using `printf`. To format a decimal integer for output, you can use the `%d` placeholder in the `printf` format string (i.e., the first parameter).

5. Write a simple "race-pace" calculator. Ask a user to type the race distance (in miles), and a string representing the time they want to finish the race in, using a format like "HH:MM:SS". Compute and return the pace per-mile required to achieve the finish time. A few notes and hints about this problem:

    - You should accept the miles value as a floating point value. You can use the standard library function `atof` to convert a string to a floating point value. Any floating point variables can be declared as either `float` or `double` (just like Java).
    - You can assume that the string entered by the user for finish time is *exactly* in the format "HH:MM:SS", for simplicity. Assume that if the user wants to finish in 31 minutes and 19 seconds, they type "00:31:19".

CHAPTER

THREE

BASIC TYPES AND OPERATORS

C provides a standard, though minimal, set of basic data types. Sometimes these are called "primitive" types. This chapter focuses on defining and showing examples of using the various types, including operators and expressions that can be used. More complex data structures can be built up from the basic types described below, as we will see in later chapters.

3.1 Integer types

There are several integer types in C, differing primarily in their bit widths and thus the range of values they can accommodate. Each integer type can also be *signed* or *unsigned*. Signed integer types have a range $-2^{width-1}..2^{width-1}-1$, and unsigned integers have a range $0..2^{width}-1$. There are five basic integer types:

`char`: One ASCII character The size of a `char` is almost always 8 bits, or 1 byte. 8 bits provides a signed range of -128..127 or an unsigned range is 0..255, which is enough to hold a single ASCII character[1]. `char` is also required to be the "smallest addressable unit" for the machine — each byte in memory has its own address.

`short`: A "small" integer A `short` is typically 16 bits, which provides a signed range of -32768..32767. It is less common to use a `short` than a `char`, `int`, or something larger.

`int`: A "default" integer size An `int` is typically 32 bits (4 bytes), though it is only guaranteed to be at least 16 bits. It is defined to be the "most comfortable" size for the computer architecture for which the compiler is targetted. If you do not really care about the range for an integer variable, declare it `int` since that is likely to be an appropriate size which works well for that machine.

`long`: A large integer A least 32 bits. On a 32-bit machine, it will usually be 32 bits, but on a 64 bit machine, it will usually be 64 bits.

`long long` Modern C compilers also support `long long` as an integer type, which is a 64-bit integer.

The integer types can be preceded by the qualifier `unsigned` which disallows representing negative numbers and doubles the largest positive number representable. For example, a 16 bit implementation of short can store numbers in the range -32768..32767, while `unsigned short` can store 0..65535.

Although it may be tempting to use `unsigned` integer types in various situations, you should generally just use signed integers unless you really need an unsigned type. Why? The main reason is that it is common to write comparisons like `x < 0`, but if `x` is unsigned, this expression *can never be true*! A good compiler will warn you in such a situation, but it's best to avoid it to begin with. So, unless you really need an unsigned type (e.g., for creating a bitfield), just use a signed type.

[1] Non-ASCII characters can also be represented in C, such as characters in Cyrillic, Hangul, Simplified Chinese, and Emoji, but not in a single 8-bit data type. See http://en.wikipedia.org/wiki/Wide_character for some information on data types to support these character types.

Integers in Python and Java compared with C

In Python, an integer can be arbitrarily large (negative or positive). Any limit on the maximum size of an int is due to available memory, not restrictions related to processor architecture. C is, of course, very much *unlike* that. Issues of overflow and underflow come into play with C, and can be very tricky to detect and debug (a sidebar below discusses the overflow issue).

Java contains (almost) the same basic integer types as in C. It has `short`, `int`, and `long`, which are 2 bytes, 4 bytes, and 8 bytes respectively (i.e., generally as they are in C). Java also has a `byte` type, which is like `char` in C: a 1-byte integer. A `char` in Java is **not** treated as an integer: it is a single Unicode character. Also, all integer types in Java are signed; unsigned integer types don't exist.

3.1.1 The `sizeof` keyword

There is a keyword in C called `sizeof` that works like a function and returns the number of bytes occupied by a type or variable. If there is ever a need to know the size of something, just use `sizeof`. Here is an example of how `sizeof` can be used to print out the sizes of the various integer types on any computer system. Note that the `%lu` format placeholder in each of the format strings to `printf` means "unsigned long integer", which is what `sizeof` returns. (As an exercise, change `%lu` to `%d` and recompile with **`clang`**. It will helpfully tell you that something is fishy with the `printf` call.)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char c = 'a';
    short s = 0xbeef;
    int i = 100000;
    long l = 100000000L;
    long long ll = 60000000000LL;
    printf("A char is %lu bytes\n", sizeof(c));
    printf("A short is %lu bytes\n", sizeof(s));
    printf("An int is %lu bytes\n", sizeof(i));
    printf("A long is %lu bytes\n", sizeof(l));
    printf("A long long is %lu bytes\n", sizeof(ll));
    return EXIT_SUCCESS;
}
```

When the above program is run on a 32-bit machine[2], the output is:

```
A char is 1 bytes
A short is 2 bytes
An int is 4 bytes
A long is 4 bytes
A long long is 8 bytes
```

and when the program is run on a 64-bit machine, the output is:

```
A char is 1 bytes
A short is 2 bytes
An int is 4 bytes
```

(continues on next page)

[2] To find out whether your machine is 64 bit or 32 bit, you can do the following. On Linux, just type `uname -p` at a terminal. If the output is `i386`, you have a 32-bit OS. If it is `x86_64`, it is 64 bits. All recent versions of MacOS X are 64 bits, so unless you're running something extremely old, you've got 64.

(continued from previous page)

```
A long is 8 bytes
A long long is 8 bytes
```

Notice that the key difference above is that on a 64-bit platform, the `long` type is 8 bytes (64 bits), but only 4 bytes (32 bits) on a 32-bit platform.

Integer sizes and source code portability

Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware. Unfortunately it occasionally leads to bugs where a program runs differently on a 32-bit machine than it runs on a 64-bit machine. In particular, if you are designing a function that will be implemented on several different machines, it is best to explicitly specify the sizes of integral types. If you `#include <stdint.h>`, you can use types that explicitly indicate their bit-widths: `int8_t`, `int16_t`, `int32_t`, and `int64_t`. There are also `unsigned` variants of these types: `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`.

For some operating systems-related functions it is extremely important to be sure that a variable is *exactly* of a given size. These types come in handy in those situations, too.

3.1.2 `char` literals

A `char` literal is written with single quotes (') like `'A'` or `'z'`. The char constant `'A'` is really just a synonym for the ordinary integer value 65, which is the ASCII value for uppercase 'A'. There are also special case `char` constants for certain characters, such as `'\t'` for tab, and `'\n'` for newline.

`'A'` Uppercase 'A' character

`'\n'` Newline character

`'\t'` Tab character

`'\0'` The "null" character — integer value 0 (totally different from the char digit '0'!). Remember that this is the special character used to terminal strings in C.

`'\012'` The character with value 12 in octal, which is decimal 10 (and corresponds to the newline character). Octal representations of chars and integers shows up here and there, but is not especially common any more.

`0x20` The character with hexadecimal value 20, which is 32 in decimal (and corresponds to the space `' '` character). Hexadecimal representations of chars and integers is fairly common in operating systems code.

3.1.3 `int` literals

Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long, such as 42L. Similarly, an integer literal may be followed by 'LL' to indicate that it is of type `long long`. Adding a 'U' before 'L' or 'LL' can be used to specify that the value is unsigned, e.g., 42ULL is an `unsigned long long` type.

An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal (base 16) — 0x10 is way of expressing the decimal number 16. Similarly, a constant may be written in octal (base 8) by preceding it with "0" — 012 is a way of expressing the decimal number 10.

Type combination and promotion

The integral types may be mixed together in arithmetic expressions since they are all basically just integers. That includes the `char` type (unlike Java, in which the `byte` type would need to be used to specify a single-byte integer). For example, `char` and `int` can be combined in arithmetic expressions such as (`'b' + 5`). How does the compiler deal with the different widths present in such an expression? In such a case, the compiler "promotes" the smaller type (`char`) to be the same size as the larger type (`int`) before combining the values. Promotions are determined at compile time based purely on the types of the values in the expressions. Promotions do not lose information — they always convert from one type to a compatible, larger type to avoid losing information. However, an assignment (or explicit cast) from a larger type to smaller type (e.g., assigning an `int` value to a `short` variable) may indeed lose information.

Pitfall: `int` overflow

Remember that wonderful algorithm called "binary search"? As an engineer at Google discovered some time ago, nearly all implementations of binary search are coded incorrectly[3]. The problem is usually on the line that computes the midpoint of an array, which often looks like this:

```
int mid = (low + high) / 2;
```

So what's the problem? The issue is that for very large arrays, the expression `low + high` may exceed the size of a 32-bit integer, resulting in "overflow", and the value resulting from an overflow is *undefined*! There is no guarantee that the high-order bit(s) will simply be truncated. In C, the result is that the array index (`mid`) overflows to an undefined value, resulting in undefined and likely incorrect program behavior. See the footnote reference ([2]) for ways to fix the code in both Java and C/C++.

3.2 Floating point types

`float` Single precision floating point number typical size: 32 bits (4 bytes)

`double` Double precision floating point number typical size: 64 bits (8 bytes)

`long double` A "quad-precision" floating point number. 128 bits on modern Linux and MacOS X machines (16 bytes). Possibly even bigger floating point number (somewhat obscure)

Constants in the source code such as 3.14 default to type `double` unless they are suffixed with an 'f' (`float`) or 'l' (`long double`). Single precision equates to about 6 digits of precision and double is about 15 digits of precision. Most C programs use `double` for their computations, since the additional precision is usually well worth the additional 4 bytes of memory usage. The only reason to use `float` is to save on memory consumption, but in normal user programs the tradeoff just isn't worth it.

The main thing to remember about floating point computations is that they are *inexact*. For example, what is the value of the following double expression?

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0) // is this equal to 1.0 exactly?
```

The sum may or may not be 1.0 exactly, and it may vary from one type of machine to another. For this reason, you should never compare floating numbers to each other for equality (==) — use inequality (<) comparisons instead. Realize that a correct C program run on different computers may produce slightly different outputs in the rightmost digits of its floating point computations.

[3] http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

3.3 Boolean type

In C prior to the C99 standard, there was no distinct Boolean type. Instead, integer values were used to indicate true or false: zero (0) means false, and anything non-zero means true. So, the following code:

```
int i = 0;
while (i - 10) {

    // ...

}
```

will execute until the variable `i` takes on the value 10 at which time the expression (i - 10) will become false (i.e., 0).

In the C99 revision, a `bool` type was added to the language, but the vast majority of existing C code uses integers as quasi-Boolean values. In C99, you must add `#include <stdbool.h>` to your code to gain access to the `bool` type. Using the C99 `bool` type, we could modify the above code to use a Boolean flag variable as follows:

```
#include <stdbool.h>

// ...

int i = 0;
bool done = false;
while (!done) {

    // ...

    done = i - 10 == 0
}
```

3.4 Basic syntactic elements

3.4.1 Comments

Comments in C are enclosed by slash/star pairs:

```
/* .. comments .. */
```

which may cross multiple lines. C++ introduced a form of comment started by two slashes and extending to the end of
// comment until the line end

The // comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

Along with well-chosen function names, comments are an important part of well written code. Comments should not just repeat what the code says. Comments should describe what the code accomplishes which is much more interesting than a translation of what each statement does. Comments should also narrate what is tricky or non-obvious about a section of code.

3.4.2 Variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. The following declares an `int` variable named "num" and the 2nd line stores the value 42 into num:

```
int num = 42;
```

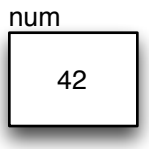


Fig. 1: A simple memory diagram for `int num = 42;`.

A variable corresponds to an area of memory which can store a value of the given type. Making a drawing is an excellent way to think about the variables in a program. Draw each variable as box with the current value inside the box. This may seem like a "newbie" technique, but when you are buried in some horribly complex programming problem, it will almost certainly help to draw things out as a way to think through the problem. Embrace your inner noob.

Initial values in variables and *undefined* values

Unlike Java, **variables in C do not have their memory cleared or set in any way when they are allocated at run time**. The value in a variable at the time it is declared is *undefined*: it is likely to be filled with what ever variable or value previously occupied that particular location in memory. Or it might be zeroes. Or it might be filled with fuzzy pink pandas. The point is that you should never assume that a variable has any value stored in it at the time of declaration. As a result, you should almost always *explicitly initialize variables* at the point of declaration. A good compiler will (usually) tell you when you're playing with fire with respect to variable initialization, but it is good to get into the habit of explicitly initializing variables to avoid this pitfall.

Undefined values in C come up in a few other places. For example, although you'd like to *think* that the following assignment results in `-128` being stored in `c`:

```
char c = 127 + 1;
```

you cannot assume that it has any particular value since the result of an overflow is *undefined*. Although the `gcc` has a special flag `-fwrapv` which forces overflow to result in two's complement wraparound, there's no guarantee of this behavior in the absence of the flag.

For lots of good discussion on undefined behaviors in C, see [Regehr] and [Lattner].

Names in C are *case sensitive* so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but may not begin with a digit. Multiple variables can be declared after the type by separating them with commas. C is a classical "compile time" language — the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter).

3.4.3 Assignment Operator =

The assignment operator is the single equals sign (=):

```
i = 6;
i = i + 1;
```

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value. Some programmers will use that feature to write things like the following:

```
y = x = 2 * x;  // double x, and also put x's new value in y
```

Demotion on assignment

The opposite of promotion, demotion moves a value from a type to a smaller type. This is a situation to be avoided, because strictly speaking the result is *implementation and compiler-defined*. In other words, there's no guarantee what will happen, and it may be different depending on the compiler used. A common behavior is for any extra bits to be truncated, but you should not depend on that. At least a good compiler (like `clang`) will generate a compile time warning in this type of situation.

The assignment of a floating point type to an integer type will *truncate* the fractional part of the number. The following code will set `i` to the value 3. This happens when assigning a floating point number to an integer or passing a floating point number to a function which takes an integer. If the integer portion of a floating point number is too big to be represented in the integer being assigned to, the result is the ghastly *undefined* (see[4]). Most modern compilers will warn about implicit conversions like in the code below, but not all.

```
int i;
i = 3.14159; // truncation of a float value to int
```

3.4.4 Arithmetic operations

C includes the usual binary and unary arithmetic operators. It is good practice to use parentheses if there is ever any question or ambiguity surrounding order of operations. The compiler will optimize the expression anyway, so as a programmer you should always strive for *maximum readability* rather than some perceived notion of what is efficient or not. The operators are sensitive to the type of the operands. So division (`/`) with two integer arguments will do integer division. If either argument is a float, it does floating point division. So (`6/4`) evaluates to 1 while (`6/4.0`) evaluates to 1.5 — the 6 is promoted to 6.0 before the division.

Operator	Meaning
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Remainder (mod)

Pitfall: int vs. float Arithmetic

Here's an example of the sort of code where `int` vs. `float` arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100:

[4] See the C11 standard: https://port70.net/~nsz/c/c11/

```
{
    int score;
    ...  // suppose score gets set in the range 0..20 somehow
    score = (score / 20) * 100;         // NO -- score/20 truncates to 0
    ...
}
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number:

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast
score = (score / 20.0) * 100;       // OK -- floating point division from 20.0
score = (int)(score / 20.0) * 100;  // NO -- the (int) truncates the floating
                                    // quotient back to 0
```

Note that these problems are similar to `int` versus `float` problems in Python (version 2). In Python 3, division using / *always* returns a floating point type, which eliminates the problem. (If integer division is desired in Python 3, the // operator can be used.)

3.4.5 Unary Increment Operators: ++ and --

The unary ++ and -- operators increment or decrement the value in a variable. There are "pre" and "post" variants for both operators which do slightly different things (explained below).

Operator	Meaning
var++	increment "post" variant
++var	increment "pre" variant
var--	decrement "post" variant
--var	decrement "pre" variant

An example using post increment/decrement:

```
int i = 42;
i++;     // increment on i
// i is now 43
i--;     // decrement on i
// i is now 42
```

Pre- and post- variations

The pre-/post- variation has to do with nesting a variable with the increment or decrement operator inside an expression — should the entire expression represent the value of the variable *before* or *after* the change? These operators can be confusing to read in code and are often best avoided, but here is an example:

```
int i = 42;
int j;
j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)
j = (++i + 10)
// i is now 44
// j is now 54
```

3.4.6 Relational Operators

These operate on integer or floating point values and return a 0 or 1 boolean value.

Operator	Meaning
==	Equal
!=	Not Equal
>	Greater Than
<	Less Than
>=	Greater or Equal
<=	Less or Equal

To see if `x` equals three, write something like `if (x==3) ...`.

pitfall: `= != ==`

An absolutely classic pitfall is to write assignment (=) when you mean comparison (==). This would not be such a problem, except the incorrect assignment version compiles fine because the compiler assumes you mean to use the value returned by the assignment. This is rarely what you want: `if (x=3) ...`.

This does not test if `x` is 3! It sets `x` to the value 3, and then returns the 3 to the `if statement` for testing. 3 is not 0, so it counts as "true" every time.

Some compilers will emit warnings for these types of expressions, but a better technique that many C programmers use to avoid such problems is to put the literal value on the *left hand side* of the expression as in: `if (3=x)...`

In this case, a compile-time error would result (you can't assign anything to a literal).

3.4.7 Logical Operators

The value 0 is false, anything else is true. The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced. (Such operators are called "short circuiting") In ANSI C, these are furthermore guaranteed to use 1 to represent true, and not just some random non-zero bit pattern. However, there are many C programs out there which use values other than 1 for true (non-zero pointers for example), so when programming, do not assume that a true boolean is necessarily 1 exactly.

Operator	Meaning
!	Boolean not (unary)
&&	Boolean and
||	Boolean or

3.4.8 Bitwise Operators

C includes operators to manipulate memory at the bit level. This is useful for writing low-level hardware or operating system code where the ordinary abstractions of numbers, characters, pointers, etc... are insufficient – an increasingly rare need. Bit manipulation code tends to be less "portable". Code is "portable" if with no programmer intervention it compiles and runs correctly on different types of processors. The bitwise operations are typically used with unsigned types. In particular, the shift operations are guaranteed to shift zeroes into the newly vacated positions when used on unsigned values.

Operator	Meaning
~	Bitwise Negation (unary) – flip 0 to 1 and 1 to 0 throughout
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or
>>	Right Shift by right hand side (RHS) (divide by power of 2)
<<	Left Shift by RHS (multiply by power of 2)

Do not confuse the bitwise operators with the logical operators. The bitwise connectives are one character wide (`&`, `|`) while the boolean connectives are two characters wide (`&&`, `||`). The bitwise operators have higher precedence than the boolean operators. The compiler will usually not help you out with a type error if you use `&` when you meant `&&`.

3.4.9 Other Assignment Operators

In addition to the plain = operator, C includes many shorthand operators which represents variations on the basic =. For example += adds the right hand side to the left hand side. `x = x + 10` can be reduced to `x += 10`. Note that these operators are much like similar operators in other languages, like Python and Java. Here is the list of assignment shorthand operators:

Operator	Meaning
`+=, -=`	Increment or decrement by RHS
`*=, /=`	Multiply or divide by RHS
`%=`	Mod by RHS
`>>=`	Bitwise right shift by RHS (divide by power of 2)
`<<=`	Bitwise left shift by RHS (multiply by power of 2)
`&=, |=, ^=`	Bitwise and, or, xor by RHS

Exercises

The theme for the following exercises is *dates and times*, which often involve lots of interesting calculations (sometimes using truncating integer arithmetic, sometimes using modular arithmetic, sometimes both), and thus good opportunities to use various types of arithmetic operations, comparisons, and assignments.

1. Write a C program that asks for a year and prints whether the year is a leap year. See the Wikipedia page on leap year for how to test whether a given year is a leap year. Study the first program in the tutorial chapter for how to collect a value from keyboard input, and use the `atoi` function to convert a C string (char array) value to an integer.
2. Write a program that asks for year, month, and day values and compute the corresponding Julian Day value. See the Wikipedia page on Julian Day for an algorithm for doing that. (See specifically the expression titled "Converting Gregorian calendar date to Julian Day Number".)
3. Extend the previous program to compute the Julian date value (a floating point value), using the computation described in "Finding Julian date given Julian day number and time of day" on the Wikipedia page linked in the previous problem. Note that you'll need to additionally ask for the current hour, minute and second from the keyboard.
4. Write a program that asks for a year value and computes and prints the month and day of Easter in that year. The Wikipedia page on Computus provides more than one algorithm for doing so. Try using the "Anonymous Gregorian algorithm" or the "Gauss algorithm", which is a personal favorite.

References

CHAPTER

FOUR

CONTROL STRUCTURES

In this chapter, we encounter the set of "control" structures in C, such as conditional statements and looping constructs. As a preview, the control structures in C are nearly identical to those found in Java (since Java syntax is heavily based on C), and bear strong resemblance to control structures found in other programming languages.

4.1 `if` Statement

Both an `if` and an `if-else` are available in C. The <expression> can be any valid C expression. The parentheses around the expression are required, even if it is just a single variable:

```
if (<expression>) <statement>    // simple form with no {}'s or else clause
```

You should **always** use curly braces around the statement block associated with an `if` statement. Although C allows a programmer *not* to include the curly braces in the case of `if` statements and other control structures when there is only a single statement in the statement block, you should **always use the curly braces**. Why, you ask, should I type those additional characters? Just ask the unfortunate engineering group at Apple that introduced the "goto fail" bug into their SSL (secure sockets layer) library: a bug that affected the macOS and iOS operating systems quite severely[1]. The upshot of this security failure is that it could have been prevented with the rigorous use of curly braces for all statement blocks.

So, for the simple form of the `if` statement shown above, you should *really* write:

```
if (<expression>) { <statement> }   // simple form with no {}'s or else clause
                                    // note the curly braces around the statement
                                    // block!
```

Code blocks use curly braces ({})

C uses curly braces ({}) to group multiple statements together, very much like Java. Whitespace is generally insignificant, very much *unlike* Python. Not surprisingly, within a code block the statements are executed in order.

Note that older versions of C (pre-C99) required that all variables be declared at the beginning of a code block. Since the C99 standard, however, variables can be declared on any line of code, as in Java and C++.

[1] See https://gotofail.com for a variety of information about the bug, and https://www.imperialviolet.org/2014/02/22/applebug.html for detailed analysis of the source code that caused the problem.

As in Java, the `else` keyword can be used to provide alternative execution for a conditional expression. Also similar to Java, multiple `if ... else if` statements can be chained together. There is no `elif` as in Python (or `elsif` as in Ruby).

```
if (<expression>) { // simple form with {}'s to group statements
    <statements>
}

if (<expression>) { // full then/else form
    <statements>
} else {
    <statements>
}

if (<expression1>) { // chained if/else if
    <statements>
} else if (<expression2>) {
    <statements>
} ...
```

4.2 The conditional expression (ternary operator)

The conditional expression can be used as a shorthand for some if-else statements. The general syntax of the conditional operator is:

```
<expression1> ? <expression2> : <expression3>
```

This is an *expression*, not a *statement*, so it represents a value. The operator works by evaluating `expression1`. If it is true (non-zero), it evaluates and returns `expression2`. Otherwise, it evaluates and returns `expression3`.

The classic example of the ternary operator is to return the smaller of two variables. Instead of writing:

```
if (x < y) {
    min = x;
} else {
    min = y;
}
```

you can write:

```
min = (x < y) ? x : y;
```

The ternary operator is viewed by some programmers as "excessively tricky" since expressions with such operators can be hard to read. Use your best judgment, and don't do something this [Horrific] example.

4.3 `switch` statement

The switch statement is a sort of specialized form of `if` with the goal of efficiently separating different blocks of code based on the value of an integer. The `switch` expression is evaluated, and then the flow of control jumps to the matching const-expression `case`. The `case` expressions are typically `int` or `char` constants (unfortunately, you cannot use strings as `case` expressions). The `switch` statement is probably the single most syntactically awkward and error-prone feature of the C language:

```
switch (<expression>) {
    case <const-expression-1>:
        <statement>
        break;
    case <const-expression-2>:
        <statement>
        break;
    case <const-expression-3>:
    case <const-expression-4>:  // here we combine case 3 and 4
        <statement>
        break;
    default:    // optional
        <statement>
}
```

Each constant needs its own `case` keyword and a trailing colon (:). Once execution has jumped to a particular case, the program will keep running through all the cases from that point down — this so called *fall through* operation is used in the above example so that `expression-3` and `expression-4` run the same statements. The explicit `break` statements are necessary to exit the `switch`. Omitting the `break` statements is a common error — it compiles, but leads to inadvertent, unexpected, and likely erroneous fall-through behavior.

Why does the `switch` statement fall-through behavior work the way it does? The best explanation might be that C was originally developed for an audience of assembly language programmers. The assembly language programmers were used to the idea of a "jump table" with fall-through behavior, so that's the way C does it (it's also relatively easy to implement it this way). Unfortunately, the audience for C is now quite different, and the fall-through behavior is widely regarded as an unfortunate part of the language.

4.4 `while` loop

The `while` loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. The conditional expression requires parentheses like the if:

```
while (<expression>) {
    <statement>
}
```

Although the curly braces are not technically required if there is only one statement in the body of the `while` loop, you should *always* use the curly braces. Again, see[1] for why.

4.5 `do-while` loop

Like a `while` loop, but with the test condition at the *bottom* of the loop. The loop body will always execute at least once. The `do-while` tends to be an unpopular area of the language. Although many users of C use the straight `while` if possible, a `do-while` loop can be very useful in some situations:

```
do {
    <statement>
} while (<expression>);
```

4.6 for loop

The for loop in C contains three components that are often used in looping constructs, making it a fairly convenient statement to use. The three parts are an initializer, a continuation condition, and an action, as in:

```
for (<initializer>; <continuation>; <action>) {
    <statement>
}
```

The initializer is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true. After every execution of the loop, the action is executed. The following example executes 10 times by counting 0..9. Many loops look very much like the following:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
    }
    return EXIT_SUCCESS;
}
```

C programs often have series of the form 0..(some_number-1). It's idiomatic in C for loops like the example above to start at 0 and use < in the test so the series runs up to but not equal to the upper bound. In other languages you might start at 1 and use <= in the test.

Each of the three parts of the for loop can be made up of multiple expressions separated by commas. Expressions separated by commas are executed in order, left to right, and represent the value of the last expression.

Note that in the C standard prior to C99, it was illegal to declare a variable in the initializer part of a for loop. In C99, however, it is perfectly legal. If you compile the above code using **gcc** without the -std=c99 flag, you will get the following error:

```
forloop.c: In function âĂŸmainâĂŹ:
forloop.c:4:5: error: âĂŸforâĂŹ loop initial declarations are only allowed in C99 mode
forloop.c:4:5: note: use option -std=c99 or -std=gnu99 to compile your code
```

Once the -std=c99 flag is added, the code compiles correctly, as expected.

4.6.1 break

The break statement causes execution to exit the current loop or switch statement. Stylistically speaking, break has the potential to be a bit vulgar. It is preferable to use a straight while with a single conditional expression at the top if possible, but sometimes you are forced to use a break because the test can occur only somewhere in the midst of the statements in the loop body. To keep the code readable, be sure to make the break obvious — forgetting to account for the action of a break is a traditional source of bugs in loop behavior:

```
while (<expression>) {
    <statement>
    statement>
    if (<condition which can only be evaluated here>) {
```

(continues on next page)

(continued from previous page)

```
        break;
    }
    <statement>
    <statement>
}
// control jumps down here on the break
```

The `break` does not work with `if`; it only works in loops and switches. Thinking that a `break` refers to an `if` when it really refers to the enclosing `while` has created some high-quality bugs. When using a `break`, it is nice to write the enclosing loop to iterate in the most straightforward, obvious, normal way, and then use the `break` to explicitly catch the exceptional, weird cases.

4.6.2 `continue`

The `continue` statement causes control to jump to the bottom of the loop, effectively skipping over any code below the `continue`. As with `break`, this has a reputation as being vulgar, so use it sparingly. You can almost always get the effect more clearly using an `if` inside your loop:

```
while (<expression>) {
    <statement>
    if (<condition>) {
        continue;
    }
    <statement>
    <statement>
    // control jumps here on the continue
}
```

4.6.3 Statement labels and `goto`

Continuing the theme of statements that have a tendency of being a bit vulgar, we come to the king of vulgarity, the infamous `goto` statement [Goto]. The structure of a `goto` statement in C is to *unconditionally* jump to a statement label, and continue execution from there. The basic structure is:

```
label:
  <statement>
  ..
  <statement>
  goto label;
```

The `goto` statement is not uncommon to encounter in operating systems code when there is a legitimate need to handle complex errors that can happen. A pattern that you might see is something like:

```
int complex_function(void) {
    if (initialize_1() != SUCCESS) { goto out1; }
    if (initialize_2() != SUCCESS) { goto out2; }
    if (initialize_3() != SUCCESS) { goto out3; }

    /* other statements */

    return SUCCESS;

  out3:
```

(continues on next page)

(continued from previous page)

```
    deinitialize_3();
  out2:
    deinitialize_2();
  out1:
    deinitialize_1();
  return ERROR;
}
```

Notice the structure above: there are multiple steps being performed to carry out some initialization for an operation[2]. If one of those initialization operations fails, code execution transfers to a statement to handle *deinitialization*, and those de-init operations happen in *reverse order of initialization*. It is possible to rewrite the above code to use `if`/`else` structures, but the structure becomes much more complex (see an exercise below). Although `goto` has the reputation of leading to "spaghetti code", judicious use of this statement in situations like the above makes for cleaner and clearer code.

Exercises

1. Rewrite the `goto` example code above (the last code example, above) to use `if`/`else` instead. Which code do you think exhibits a more clear structure?
2. Consider the following program snippet:

   ```
   char buffer[64];
   printf("Enter an integer: ");
   fgets(buffer, 64, stdin);
   int val = atoi(buffer); // convert the string to an integer
   if (val % 2 == 1)
     val *= 2;
     val += 1
   printf("%d\n", val);
   ```

   What is printed if the number 3 is entered?
3. Say that you want to write a program that repeatedly asks for a snowfall amount, and that you want to keep asking for another value until the sum of all values exceeds a certain value. What control structure would work best to facilitate entry of the snowfall values, and why?
4. Say you want to write a program that computes the average of quiz scores. You have a big stack of quizzes, so you do not know the number of quizzes up front. What control structure would work best to facilitate entry of the scores, and why?
5. Say that you want to simulate rolling a die (singular of dice) a fixed number of times and to compute and print the average value for the die rolls. What control structure would work best for this problem, and why?

References

[2] This example was adapted from https://blog.regehr.org/archives/894, where you can find additional discussion on tasteful use of `goto` in systems code.

CHAPTER

FIVE

ARRAYS AND STRINGS

5.1 Arrays

Arrays in C are declared and used much like they are in Java. The syntax for using arrays in C is nearly identical to Java, and is very similar to the syntax used for Python lists. As with both Python and Java, arrays in C are always indexed starting at 0. Thus, in the following example, the first `int` in the `scores` array is `scores[0]` and the last is `scores[99]`:

```
1 int scores[100];
2 scores[0]  = 13;          // set first element
3 scores[99] = 42;          // set last element
```

The name of the array refers, in some sense, to the *whole array* but in actuality, the array name refers to the *memory address* at which the array storage begins. As in Java, elements of an array in C are stored *contiguously*. Thus, for the above array, if the first element in the array is stored at memory address *x*, the next element is stored at *x+4* (since the `int` is 4 bytes on most machines today), as depicted in *A memory diagram of the scores array.*, below.

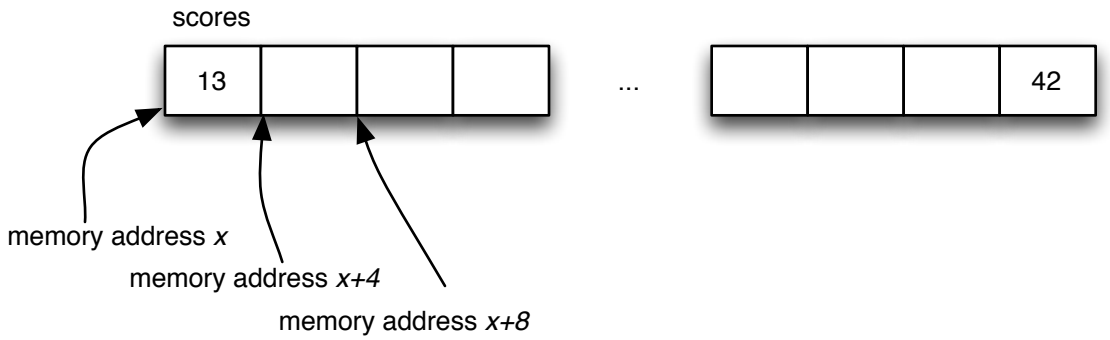


Fig. 1: A memory diagram of the scores array.

5.1.1 Array initialization

Note that because C does not do any automatic initialization of variables, the array has *undefined* contents at the point of declaration (line 1, above). A common practice is to use either a simple `for` loop construct to set all values in the array to a specific value, e.g.,:

```
int scores[100];
for (int i = 0; i < 100; i++) {
```

(continues on next page)

(continued from previous page)

```
    scores[i] = 0;
}
```

Another common practice is to use the `memset` function or `bzero` function to set everything in an array to zeroes. The `memset` function is declared in `strings.h` (so you need to `#include` it), and takes three parameters: a memory address (which can just be the name of the array), the character that should be written into each byte of the memory address, and the number of bytes to set. Thus, the above `for` loop could be replaced with the following:

```
// at the top of your source code
#include <string.h>

int scores[100];
memset(scores, 0, 100*sizeof(int));
```

Note that we need to specify the number of *bytes* we want to set to 0, thus we say `sizeof(int)` multiplied by the number of array elements. It's always good practice to use `sizeof`, even if you think you can assume that the size of an `int` is 4. Don't make that assumption; use `sizeof`.

One last way that array contents can be initialized is to use C initializer syntax. Say that we just want to create an array of 10 scores. We could initialize the array as follows:

```
int scores[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The initializer syntax in C is just a set of curly braces, within which are comma-separate values. You can even leave off the array size if you give an initializer, and the compiler will figure out how large to make the array:

```
int scores[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // equivalent to above
```

The initializer syntax is especially useful for small-ish arrays for which the initial values are not all identical.

5.1.2 `sizeof` and arrays

The built-in `sizeof` function works with arrays. Specifically, it will return the number of bytes occupied by the array, which is quite helpful. So, the `memset` code we wrote earlier could be replaced with:

```
// at the top of your source code
#include <string.h>

int scores[100];
memset(scores, 0, sizeof(scores));
```

5.1.3 No array bounds checking!

It is a very common error to try to refer to a non-existent array element. Unlike Java or Python, in which an out-of-bounds array (or list) reference will result in an exception, C will happily attempt to access the non-existent element. The program behavior in such a case is *undefined*, which basically means that anything can happen. Your program might crash, but it might not. It might behave as you expect, but it might not. It might cause your computer to levitate or to spontaneously combust. Who knows? Yuck. Welcome to C.

So what you can you do about this? The best thing is to use good tools for detecting memory corruption and bad array accesses. The **valgrind** tool[1] is especially good in this regard, and is highly recommended. Its output can be somewhat difficult to understand at first, but it is a hugely helpful tool when trying to debug seemingly random program behavior.

Besides **valgrind**, you can use the **clang** *static analyzer*. This tool analyzes your code to find potential bugs, too, but it is pretty fast (it doesn't actually execute your code) and the output is a little easier to grasp than **valgrind**. The tool to invoke is called **scan-build**[2], and can be used on the command line *before* any compiler tols that you invoke. For example, consider the following program that increments an uninitialized variable (thus leading to undefined behavior):

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int x;
    x += 10;
    printf("%d\n", x);
    return EXIT_SUCCESS;
}
```

Running **scan-build** on this code results in the following:

```
$ scan-build clang uninit.c
scan-build: Using '/usr/lib/llvm-3.8/bin/clang' for static analysis
uninit.c:6:7: warning: The left expression of the compound assignment is an uninitialized value.␣
↪The computed value will also be garbage
    x += 10;
    ~ ^
1 warning generated.
scan-build: 1 bug found.
```

This is helpful! It's something that ordinary compilation will *not* uncover (**clang** compiles this program without warning or error, ordinarily), and errors like this are pretty easy to make for inexperienced C programmers, especially when it comes to arrays and pointers (a topic coming soon). Advice: run **scan-build** as part of your C regimen.

5.1.4 Variable length arrays

At the point of declaration, the size of an array in C *can* be specified with a variable, which creates what is called a *variable length array*. Variable length arrays were added to C in the C99 standard, so if you use a variable when specifying the size of an array and there is a compile-time error on that line, make sure that you are compiling in C99 mode (`-std=c99` on **clang** and **gcc**). Here is an example with using a variable-length array (notice that we're using the `atoi` function to convert a string to an integer):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("How many scores to input? ");
6      char buffer[10];
7      fgets(buffer, 10, stdin);
8      int num_scores = atoi(buffer);
```

(continues on next page)

[1] http://valgrind.org
[2] https://clang-analyzer.llvm.org/scan-build.html

(continued from previous page)

```
    int scores[num_scores];
    for (int i = 0; i < num_scores; i++) {
        printf("Enter score %d: ", i+1);
        fgets(buffer, 10, stdin);
        scores[i] = atoi(buffer);
    }

    // ... do something with the scores

    return EXIT_SUCCESS;
}
```

5.2 Multidimensional Arrays

Just as in Java, C allows a programmer to declare "multi-dimensional" arrays by using multiple pairs of square braces in the array variable declaration. For example, a 2-dimensional array of integers with `r` rows and `c` columns would be declared as `int array[r][c]`. Thus, if we wanted to declare a 3x3 array to hold the contents of a tic-tac-toe board, we might do something like this:

```
char board[3][3];
```

You can use array initialization syntax with multi-dimensional arrays, too. For example, the `board` declaration could set each element as follows:

```
char board[3][3] = {{'O', 'O', ' '},
                    {'X', 'X', 'O'},
                    {' ', 'O', 'X'}};
```

Note that each nested set of curly braces in the initializer refers to a *row* in the array.

The underlying implementation of a multi-dimensional array stores all the elements in a *single contiguous block of memory*. The array is arranged with the elements of the rightmost index next to each other. In other words, board[1][8] comes right before board[1][9] in memory. (This arrangement is called "row-major order"[3].)

Memory access efficiency.

If you know about CPU caches and cache lines, you'll know that it's more efficient to access memory which is near other recently accessed memory. This means that the most efficient way to read through a chunk of the array is to vary the rightmost index the most frequently since that will access elements that are near each other in memory.

5.3 C Strings

C has minimal support of character strings. A string in C is, in essence, an array of `char`s. C includes a standard library of functions for performing a variety of string operations, but the programmer is ultimately responsible for managing the underlying array (and memory) used by the string. Computations

[3] http://en.wikipedia.org/wiki/Row-major_order

involving strings is very common, so becoming a competent C programmer requires a level of adeptness at understanding and manipulating C strings.

A C string is just an array of `char` with the one additional convention that a "null" character (`'\0'`) is stored after the last character in the array, as an end-of-string marker. For example, the following code creates and prints the C string `"go 'gate"` (using some array initialization syntax introduced above):

```
char string[] = {'g', 'o', ' ', '\'', 'g', 'a', 't', 'e', '\0' };
printf("%s\n", string);
```

Notice a few things about the above line of code: (1) we don't need to specify the size of the array (the compiler can figure that out), (2) we need to "escape" the apostrophe (the 4th character), since we need to distinguish it from the character delimiters, and (3) we need to explicitly specify the end-of-string marker (null character).

Another way to initialize a C string is to use double-quotes. The following code is identical to above:

```
char string[] = "go 'gate";
printf("%s\n", string);
```

The compiler *automatically* adds the null termination character as the last character in `string`, giving an identical in-memory representation as the previous code example.

Since a C string is just an array of `char`, it is totally *mutable* (which should be, hopefully, an obvious point). As a result, we can tamper directly with the contents of the array to change the string. For example, building on the last example, we could write:

```
string[3] = 's';
string[4] = 'k';
printf("%s\n", string);
```

to change the string to `"go skate"` and print it.

5.3.1 Getting the length of a string

It is often necessary in programs to obtain the length of a string. There is a built-in `strlen` function just for this purpose. `strlen` takes a single C string as a parameter, and returns an `size_t` type, which is typically the same size as a `long` (either 4 or 8 bytes, depending whether you're on a 32-bit or 64-bit machine, respectively). `strlen` is declared in the `string.h` header file, so don't forget to include that file when using any built-in string functions like `strlen`.

Here's a brief example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char name[128];
    printf("Please type your name: ");
    fgets(name, 128, stdin);
    printf("Your name is %d characters long.\n", strlen(name)-1);
        // why strlen(name) - 1?
        // fgets includes the \n (newline) character that the user types in
        // the string filled in name, and we don't want to include that
        // character as part of the length of the name.
```

(continues on next page)

(continued from previous page)

```
    return EXIT_SUCCESS;
}
```

Which header file do I need to include?

For pretty much *all* C programs you write you will need to `#include` some header files (headers are discussed in more detail in *Program structure and compilation*). Which header file will you need? One of the easiest ways to find out is to use the `man` program in a Linux (or *NIX) shell to read the manual page for a particular C library function. For example, if you need to find out what include file to use for the function `atoi`, you could simply type `man atoi` at the command line. At the top of the man page the appropriate `#include` line will be listed. You can also use a search engine and search for `atoi man page` and *usually* you'll get the same results, but different C library versions and compilers may use slightly different header files so its best just to use the `man` pages on your system.

Manual pages can be a little bit difficult to wade through, but they are almost always divided into useful sections so you can (sort of) quickly find what you're looking for. For finding out what header file to include, look in a section at the top of the man page called "SYNOPSIS". That section also contains the function "prototypes" (which we'll discuss in a later chapter on functions), which provides the data types of any parameters and return values.

To navigate a man page, you can usually type `'d'` to go down a page, `'u'` to go up a page, and `'q'` to quit (type `'h'` or `'?'` for help on navigating). One confusing aspect of looking up a `man` page is that the same *name* can appear in multiple *sections* of the manual pages system. For example, there's a `printf` C library function, and there is also a `printf` function available for writing shell scripts. If you just type `man printf`, you'll get the shell command reference, which may not be what you want. To get the right man page, you can type `man <sectionnumber> <symbol>`, as in `man 3 printf` (the C library function `printf` is in manual section 3). To find out the manual section number, you can search the manual pages by typing `man -k printf`, which will give a list of all man pages that contain the string `printf`. The section number is shown in parens after the function name.

5.3.2 Copying strings

Recall that an array variable really just holds the memory address of the beginning of the array. Thus, = (direct assignment) cannot be used to copy strings. Instead, the characters must be copied one-by-one from one string to another. Fortunately, the pain of doing this is (somewhat) alleviated by a number of built-in C library functions to do the work for us. The best function to use for copying strings is called `strlcpy`, which takes three parameters: the destination string buffer, the source string, and the size (number of bytes) in the destination string buffer. For example, if we wanted to make a copy of a string that a user typed in, we could do the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char buffer[128];
    printf("Gimme a string: ");
    fgets(buffer, 128, stdin);
    size_t size = strlen(buffer)+1; // add 1 for the null termination character!
    char copy[size];
    strlcpy(copy, buffer, size);
```

(continues on next page)

(continued from previous page)

```
    return EXIT_SUCCESS;
}
```

Why, you may ask, do we need to pass the size of the destination string buffer as the third argument? Can't the compiler figure it out? Sadly, it cannot, especially in the case of variable length arrays and pointers (which we will encounter in a later chapter). There is an "older" C library function called `strcpy` which only takes two parameters: the destination and the source strings. One seriously bad thing that can happen with `strcpy` is exemplified by the following code:

```
char source[] = "this is a fairly long string, isn't it?";
char dest[8];  // this is a rather small buffer, isn't it?
strcpy(dest, source);
```

The `strcpy` function will happily copy the string referred to by `source` into the string referred to by `dest`. That's bad. The length of `source` is *way* longer than `dest`, so what happens is a *buffer overflow*. That is, the `strcpy` function ends up blowing past the end of the 8 bytes we allocated to `dest`, and starts writing data into what ever comes next (which happens to be on the stack of what ever function is executing). Again, clearly bad stuff. Even worse, the program may crash ... or it might not. It's impossible to tell from the source code, because the behavior (according to the C language specification) is *undefined*[4]. The moral of the story: always use `strlcpy`. Also, it may be useful to note that **`scan-build`**, described above, detects and prints a warning about this buffer overflow.

5.3.3 Comparing strings

Just as = cannot be used to copy strings, == cannot be used to compare strings. The reason is very similar to why == cannot be used in the Java language to compare strings: the comparison for equality will just compare string *references* (or "pointers", which we will encounter soon) instead of comparing the *contents* of the strings.

There are four C library functions that are commonly used to compare two strings.

`strcmp(s1, s2)` Compare C strings referred to by parameters s1 and s2. Return 0 if the string contents are equivalent, -1 if s1 lexicographically precedes s2, and 1 if s2 lexicographically precedes s1.

`strcasecmp(s1, s2)` Same as `strcmp`, but compare the strings in a case-insensitive manner.

`strncmp(s1, s2, n)` Same as `strcmp`, but only compare the first `n` characters of the two strings. (Technically `strncmp` only compares the first min(n, strlen(s1), strlen(s2)) characters).

`strncasecmp(s1, s2, n)` Same as `strncmp`, but compare the strings in a case-insensitive manner.

5.3.4 Another example

Let's look at one more example of a string manipulation program. In this program, we ask the user for a string, then convert all characters in the string to lowercase.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <ctype.h>
5
6 int main() {
```

(continues on next page)

[4] http://en.wikipedia.org/wiki/Undefined_behavior

(continued from previous page)

```
    char buffer[64];
    printf ("Gimme a string: ");
    fgets(buffer, 64, stdin);
    for (int i = 0; i < strlen(buffer); i++) {
        if (isupper(buffer[i])) {
            buffer[i] = tolower(buffer[i]);
        }
    }
    printf ("Here's your string, lower-cased: %s\n", buffer);
    return EXIT_SUCCESS;
}
```

An example run of the program might look like this:

```
Gimme a string: AbCDERX!!! whY?!
Here's your string, lower-cased: abcderx!!! why?!
```

The core of the function is a for loop that iterates through all indexes of the string, checking whether each character should be lower-cased. The code above also demonstrates a couple functions from the `#include <ctype.h>` header file (`isupper` and `tolower`). The `isupper` test (line 10) is, strictly speaking, unnecessary; calling `tolower` on an already-lowercased letter still results in a lowercase letter. Otherwise, those two new functions behave as one might expect: given a character, they return either a new character, or a (quasi-)Boolean value[5].

There are quite a few functions defined in `ctype.h`. On MacOS X you can type **`man ctype`** to get a list of those functions, and on Linux, you can type **`man islower`** (or **`man <any ctype function>`**) to get a catalog of all the various functions in `ctype.h`. The following is an incomplete list; see **`man`** pages for gory details:

- `isalnum`
- `isalpha`
- `isdigit`
- `ishexnumber`
- `islower`
- `isnumber`
- `isprint`
- `ispunct`
- `isspace`
- `isupper`
- `tolower`
- `toupper`

Exercises

1. Run the following program, which has a bad array index. What is its behavior? What if you change the `for` loop so that the second part of the for loop reads `i <= max*100` — what happens then?

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int max = 10;
    int array[max] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
```

(continues on next page)

[5] The `isupper` function returns an `int`, not `bool`, which is fairly common in C. Since the `bool` type didn't get added to the language until fairly recently, most predicate functions return an integer representing True (1) or False (0).

(continued from previous page)

```
7
8       for (int i = 0; i <= max; i++) {
9           printf("Array index %d contains %d\n", i, array[i]);
10      }
11      return EXIT_SUCCESS;
12  }
```

2. Write some code that computes the length of a string without using the built-in `strlen` function. (Defining new functions is described in a later chapter, but with some Java and/or Python knowledge, you can probably make a good guess at how to define a new function in C.)
3. Implement your own version of `strlcpy`. Instead of calling `strlcpy` on the second to last line of the *strlcpy example above*, write your own `for` loop (or some other kind of loop) to accomplish the same thing.
4. Write a program that asks for a string from a user and "strips" all whitespace characters from the end of the string (spaces, tabs and newlines). To do that, you can simply assign a null character to the character array index that follows the last non-whitespace character of the string.
5. Write a program that asks for a string from a user and prints the string in reverse.
6. Write a program that asks for a string from a user and prints whether the string is a palindrome. Don't implement this problem recursively; check the characters within the string in some type of loop structure. In your implementation, ignore non-letters and treating the string in a case-insensitive manner. For example, "A man, a plan, a canal, Panama!" should be considered a valid palindrome.
7. Write a program that asks for two strings and prints whether the two strings are anagrams of each other. This is somewhat challenging to do given what has been covered in C thus far, but good practice!

CHAPTER

SIX

AGGREGATE DATA STRUCTURES

6.1 The C `struct`

C has a facility for grouping data elements together in the form of a "record", which is called a `struct`. A `struct` in C is sort of like a class (in languages with classes), except that (1) all members of the `struct` are public (i.e., there is no way to "hide" members), and (2) there are no methods, only data members. Members of a `struct` are often referred to as `fields`.

The following code defines a type called `struct fraction` that has two integer fields named `numerator` and `denominator`. Note that a semicolon is required after the final curly brace of the declaration, as well as after the declaration of each field.

```
struct fraction {
    int numerator;
    int denominator;
};  // don't forget the semicolon!
```

Note that the full name of the new type is `struct fraction`, not just `fraction`. Thus, if we want to create a new fraction variable, we write:

```
struct fraction f1;
```

C uses the period (`.`) as the operator to access individual fields in a `struct` (similar to the way that the period is used to access instance variables in a Java or Python object). The following code assigns values to the two fields of our new variable, then prints out the contents of the `struct`:

```
f1.numerator = 3;
f1.denominator = 5;
printf("f1 is %d/%d\n", f1.numerator, f1.denominator);
```

6.1.1 Initializing `structs`

A syntax similar to array initialization can be used to initialize fields of a `struct`. For example, to declare and initialize a new `struct fraction`, we could use the following:

```
struct fraction f2 = { 3, 5};
```

which would assign 3 to the `numerator` field and 5 to the `denominator` field. An explicit field assignment syntax can also be used:

```
struct fraction f3 = { .denominator = 5, .numerator = 3 };
```

Note that using the *explicit* field assignment syntax, the field assignments do not need to appear in the same order as the original declaration of the `struct`.

Annoyance: == does not work with structs

Unfortunately, the comparison for equality operator (==) does *not* work with `structs`. In fact, the code `if (f5 == f4)` will not even compile. One way to address comparing structures is to write a function to compare them, which we will discuss in the *Functions* chapter. Another way is to directly compare the contents of memory occupied by two `structs`, which we will discuss in the *Pointers and more arrays* chapter.

6.1.2 Copying `structs`

Conveniently, the = (assignment) operator can be used to copy the contents of one struct into another. The copy is done in a field-by-field manner:

```
struct fraction f4 = { 2, 7 };
struct fraction f5 = f4;  // f5 now has identical contents as f4
```

There is another way to copy `structs` (using the built-in `memcpy` function), but since this method requires use of "pointers" we will defer discussion until the *Pointers and more arrays* chapter.

6.1.3 Arrays of `structs` and type aliases (`typedef`)

It is perfectly valid, and quite common, to have arrays of records. For example, we might want to have a whole series of fractions stored in an array, as follows:

```
1 struct fraction numbers[100];
2 numbers[0].numerator = 22;       // set the 0th struct fraction
3 numbers[0].denominator = 7;
```

The declaration on line 1, above, is a little bit of a mouthful, but reading from right-to-left can help: "an array of 100 elements called numbers, where each element contains a struct fraction". To help simplify reading and writing type complex declarations, C contains a mechanism for creating *type aliases*, using the keyword `typedef`. The syntax goes `typedef <original type> <type alias>`, as follows:

```
1 typedef struct fraction fraction_t;
2 fraction_t numbers2[100];
```

Line 1 defines a type alias for `struct fraction` called `fraction_t` (a "fraction type"). Now, `fraction_t` can be used where ever we might originally have used `struct fraction`. On line 2, an array of these fraction structures is created, which is a tiny bit easier to read than the first array declaration.

6.1.4 Using `sizeof` with a `struct` and memory layout of a `struct`

The built-in `sizeof` function works quite happily with a `struct`. It returns the number of bytes occupied by the struct in memory. As with other data types, either a variable name or a type name may be used as the argument to `sizeof`. For example, consider the following code:

```
struct fraction f6 = { 1, 2};
printf("Size of f6: %lu\n", sizeof(f6));
printf("Size of struct fraction: %lu\n", sizeof(struct fraction));
```

On almost all machines today, the output of the above code will be:

```
Size of f6: 8
Size of struct fraction: 8
```

since the size of a single int is almost always 4 bytes.

No surprises there, right? Let's look at the following program, which defines a `struct student` containing a name, class year, and age.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct student {
5      char name[32];
6      short class_year;
7      char age;
8  };
9
10 int main() {
11     struct student s = { "H. Sommers", 2026, 5 };
12     printf("An example student: %s, %d, %d\n", s.name, s.class_year, s.age);
13     printf("Size of a student struct: %lu\n", sizeof(struct student));
14     return EXIT_SUCCESS;
15 }
```

Compiling and running this code gives this output:

```
An example student: H. Sommers, 2026, 5
Size of a student struct: 36
```

Consider the size reported by the program, 36, and remember that a `short` is 2 bytes and `char` is 1 byte. Last time I checked, 32 + 2 + 1 = 35! What's happening?!

When the compiler allocates memory on the stack or the heap for a `struct`, it may introduce "padding" bytes to ensure that the entire `struct` fits within an even multiple of machine words. If the word size is 4 bytes, then the compiler will silently add `sizeof(struct) % 4` bytes as "padding" to the end of the struct[1]. So, in the `struct student` definition starting on line 3, above, there is one extra byte added by the compiler to make the entire structure occupy a "word-aligned" number of bytes. A picture of how an actual `struct student` looks in memory is thus like the following:

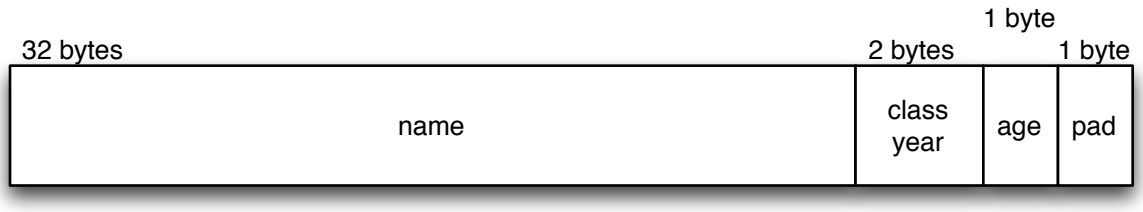


Fig. 1: An struct that has "padding" inserted by the compiler.

[1] http://en.wikipedia.org/wiki/Data_structure_alignment

The padding inserted by the compiler is not usually something one needs to pay close attention to, but in certain circumstances it *does* matter and it's good to be aware of this behavior.

Exercises

1. Assume that you have a text file with a series of student names, class years, and ages listed, like the following:

```
Alice Z., 2020, 17
Bob Y., 2019, 19
Chelsea X., 2020, 18
Draco M., 2019, 20
```

   Write a program that reads the text file contents from standard input (hint below) and stores each student in a C struct in an array. After you've loaded the students, print each of them out on a separate line, and print the average age (as a floating point number) at the end. (To format a floating point number for output using `printf`, you can use the `%f` placeholder.)

   You can assume any reasonable upper-bound for the number of characters in a name, and any reasonable upper-bound for the number of students. That is, you should *overallocate* space required for the name and number of students, within reason. An upper bound for student name might be 64, and an upper bound for the number of students might be 100.

   You can use the `fgets` call to read data from standard input (just as you've already done for keyboard input), and use "shell redirection" to cause the contents of a text file to be treated as stdin to your program. Say that you've compiled the code to an executable called `sreader` and the student data is in the text file `students.txt`, you could do the redirection trick by typing:

```
$ ./sreader < students.txt
```

2. Extend the above program to do the following. After all the student records are loaded, repeatedly ask for a student first name until the special string `DONE` is entered. For each valid name entered, search for the name in the array of records and print the values within the relevant records found (note that more than one student may match the query). If no student is found matching the name, print a message to that effect. You should allow the search (name) entered to be compared in a case-insensitive way. For example, the string "Bob" should match the 2nd record shown above. As another example, the string "E" should match both "Alice Z." and "Chelsea X." (both have "e"'s, but the other two names do not). Consider using the built-in `strcasestr` function[2] to compare strings.
3. Write a program that asks for values for two fractions (numerator and denominator for each), and computes and prints the sum of the fractions. Store the result of the sum in a new `struct fraction` prior to printing the sum.

[2] http://man7.org/linux/man-pages/man3/strstr.3.html

CHAPTER

SEVEN

FUNCTIONS

All programming languages have built-in mechanisms for *structuring* and *modularizing* the code. The main mechanism that C provides is the *subroutine*, or *function*. In fact, C provides little beyond this basic technique[1].

Good program design in C (and many other programming languages) involves creating what are typically short functions that accomplish one task, and in which there is little or no duplication of functionality. The main reasons for creating short, single-purpose functions is to make them more easily testable and to make them easier to read. There are many benefits to having one place in the code where each major component is implemented, including making it easier to modify the code and making it easier to test. These ideas are so important in programming that they are present in many different design principles, such as the Abstraction Principle[2], the Don't Repeat Yourself (DRY) principle[3], and structured programming[4].

The Abstraction Principle

"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

Benjamin C. Pierce, "Types and Programming Languages" (http://www.cis.upenn.edu/~bcpierce/tapl/).

7.1 Function syntax

A function has a name, a comma-separated list of parameters, the block of code it executes when called, and, optionally, a return value. The basic function definition syntax is:

```
return-type function-name(parameters) { code-block }
```

For example, here is a function that computes and returns *n*! (n factorial):

```
1 /*
2  * iteratively computes and returns n!
3  * if n < 0, returns 0.
```

(continues on next page)

[1] There are advanced techniques that build upon the basic mechanisms available in C to, for example, mimic capabilities found in object-oriented programming languages. As a introductory text, this book will not go into any of those techniques. One additional technique we cover in this book is found in the chapter on *Program structure and compilation*, in which we discuss a technique that provides a type of information hiding by enabling functions to remain "hidden" on a per-file basis.

[2] http://en.wikipedia.org/wiki/Abstraction_principle_(programming)

[3] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[4] http://en.wikipedia.org/wiki/Structured_programming

(continued from previous page)

```
 */
int factorial(int n) {
    if (n < 0) {
        return 0;
    }
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

We have seen most of this syntax already (with the `main` function), but it is worth reviewing.

1. On line 5, the type declaration of the function is given, and shows that the data type of the return value of the function is `int`, and that the function takes a single `int` parameter, named `n`. The function is, of course, named `factorial`.
2. There can be any number of parameters to a function (though a small number of parameters is strongly preferable). Each parameter is separated by a comma, and follows the basic syntax of a variable declaration, which is: `type-name variable-name`.
3. Any parameters to the function are *passed by value*. This means that the function receives *copies* of the arguments passed to it by the caller. If any modifications are made to those copies, they have *zero effect* on the variables or values passed to the function — any changes are confined (*local*) to the function. We will have more to say about pass-by-value function call semantics, below.
4. The `return` statement delivers a value from the function back to the caller of the function. There are `return` statements on lines 7 and 13 of this function. The first `return` handles the special case where `n` is less than 0, thus $n!$ is undefined.
5. Any variables declared inside the function are *local* to the function, just as with Java, Python, and many other programming languages. Thus, the parameter variable `n` and the local variable `result` only exist while the function is being executed.

To call the `factorial` function, a programmer uses parentheses after the function name, passing any required arguments between the parens:

```
#include <stdio.h>
#include <stdlib.h>

// ...

char buffer[8];
printf("I'll compute n! for you.  What should n be? ");
fgets(buffer, 8, stdin);
n = atoi(buffer);
int result = factorial(n);
printf ("%d! is %d\n", n, result);
```

So far, none of this should be particularly surprising. You may have already seen "public static methods" in Java (e.g., `main`!), which are very similar to C functions, or you may have already seen functions in Python (defined using the `def` keyword). Both public static methods in Java and functions in Python behave very similarly to functions in C. In fact, all of these languages use pass-by-value semantics for parameters.

7.1.1 `main` is where it all begins

Every C program **must** have a `main` function. An attempt to compile a program in C which does not have a `main` function defined somewhere will result in an error. Unlike Java, where any number of class definitions can have a `public static void main` definition, it's a highlander situation in C: *there can be only one*[5].

7.1.2 Function naming restrictions and conventions

The only *requirement* for naming C functions is similar to many programming languages: function names must either begin with a letter or underscore, and can only include numbers, letters, and underscores. Conventionally, function names that start with an underscore typically mean that the function should be treated as a "private" library function, off limits to other programmers.

As far as naming conventions, there are a wide variety of practices in existence. Some programmers like to name their functions using `lowerCamelCase`, which is common in languages such as Java, or using `snake_case`, which is common in Python and Ruby. In the C standard library, a common practice is to use short, abbreviated names consisting of a single word (e.g., `tolower`). Still others like to use the abomination referred to as Hungarian Notation[6]. A fairly widely used convention in C is `snake_case`, which is the practice followed in this book.

7.2 Data types for parameters and return values

There are, technically speaking, no restrictions on the data types of parameters or return values for functions in C. Functions in C can accept all the basic data types as parameters, as well as `struct` types, arrays, and as we will see soon, memory pointers to any data type.

Likewise, there are no syntactic restrictions on the data type of the return value from a function. C does not permit *multiple* return values, unlike some other languages, but it is permissible to return a `struct` type that contains multiple fields (or, as we will later see, a pointer to a memory block containing multiple data items).

C-ing and Nothingness — `void`

`void` is a type formalized in ANSI C which means "nothing". To indicate that a function does not return anything, use `void` as the return type. If a function does not take any parameters, its parameter list may either be empty (i.e., `()`), or it can contain the keyword `void` to indicate that the function does not take parameters. It is more common and conventional in C to use an empty parameter list for functions that don't take parameters.

7.2.1 Parameters to functions are passed by value

The key thing to remember for function parameters is that they are *passed by value*. (Note that Java also uses pass-by-value semantics for method parameters.) Passing by value means that the actual parameter values are *copied* into local storage (on the stack). This scheme is fine for many purposes, but it has two disadvantages:

1. Because the function invocation ("callee") has its own copy (or copies) of parameters, modifications to that memory are always *local to the function*. Therefore, value parameters do not allow the callee to

[5] http://en.wikipedia.org/wiki/Highlander_(film)
[6] http://en.wikipedia.org/wiki/Hungarian_notation

communicate back to the caller. The function's return value can communicate some information back to the caller, but not all problems can be solved with the single return value.

"Wait!", you may exclaim. "Can't I pass in a list variable in Python or some object variable in Java and *modify* that variable within the function or method I call?" The answer is not really. While it's true that you can modify a list that's passed into a function in Python, the parameter variable in the function really just receives a *copy of a reference to the list*, not the list itself. Same thing for Java: when you pass an object into a method, you're really *passing a reference to an object* into the method. The method receives a *copy* of the reference, allowing you to manipulate that object. We can get similar behavior with passing "pointers" into functions in C, which we'll see in the next chapter.

2. Sometimes it is undesirable to copy the value from the caller to the callee because the value is large and copying is expensive, or because, for some reason, copying the value is undesirable.

7.2.2 Example 1: an anti-example of swapping two values

As mentioned above, a key implication of pass-by-value function parameters is that the function gets *local* copies of any parameter values. Say that we want to exchange two values via a function. For example, we want to swap the numerator and denominator in a fraction `struct`. This is some code that would *not* work:

```
#include <stdlib.h>

typedef struct fraction {
    int numerator;
    int denominator;
} fraction_t;

void swap_numerator_denominator1(fraction_t frac) {
    int tmp = frac.numerator;
    frac.numerator = frac.denominator;
    frac.denominator = tmp;
}

void swap_numerator_denominator2(int nancy, int donkey) {
    int tmp = nancy;
    nancy = donkey;
    donkey = nancy;
}

int main() {
    fraction_t f1 = { 1, 3};
    swap_numerator_denominator1(f1);  // swap?  uh, no.
    swap_numerator_denominator2(f1.numerator, f1.denominator);  // no, again
    return EXIT_SUCCESS;
}
```

Epic fail, times 2. For each of the swap functions, the exchange happens only *inside* the function; the caller will *never* see any swap of nancy and donkey. The fact that the function takes a `struct` here is irrelevant for attempt 1; even if we wrote a `swap` to take two `int` parameters (i.e., attempt 2), there is no way this is going to happen. Sorry. In the next chapter on *Pointers and more arrays*, we will solve this problem.

7.2.3 Example 2: passing a `struct` to and from a function

Ok, enough of the anti-examples. Here is an example of passing *and* returning a `struct`. We'll write a function to add two fractions together and return a new fraction. A few things to note about the code below:

- The computation of the greatest common divisor is recursive.
- The use of abs in the least common multiple function requires `#include <stdlib.h>` at the top of the source code.
- The `add_fractions` function separately computes the denominator and numerator of the result of the addition, then just constructs and returns a new `fraction_t`. For both the `fraction_t` parameters *and* the return value, entire copies are made to get the arguments "in" to the function and to get the result "out".

```
// compute the greatest common divisor, recursive style
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}

// compute the least common multiple
int lcm(int a, int b) {
    return abs(a * b) / gcd(a, b);
}

// add a couple fractions together
fraction_t add_fractions(fraction_t f1, fraction_t f2) {
    int denom = lcm(f1.denominator, f2.denominator);
    int numer = (denom / f1.denominator) * f1.numerator +
                (denom / f2.denominator) * f2.numerator;
    fraction_t result = { numer, denom };
    return result;
}
```

7.2.4 Example 3: passing an array to a function

Passing an array parameter to a function is somewhat different in nature than the other parameter data types we've seen:

1. It is often the case that it is not possible to know the correct array length when declaring the *formal* parameter in the function declaration. This is actually a *good* thing in disguise: it forces us to write a more general function instead of one that specifies an array of a certain size.

   For example, say we want to write a function to multiply several fractions together, where each fraction is an element of an array. We want to write the function so that it can handle *any* array size. We write it as shown below. Notice that we leave the array size blank in the formal parameter, and pass a second parameter that specifies the number of array elements we should examine. Since an array in C does not know its own size, we are forced to pass its size separately.

```
fraction_t multiply_fractions(fraction_t fraclist[], int num_fractions) {
    fraction_t result = { 1, 1};
    for (int i = 0; i < num_fractions; i++) {
        result.numerator *= fraclist[i].numerator;
        result.denominator *= fraclist[i].denominator;
    }
    return result;
}
```

2. The second issue that makes array parameters somewhat different than any other data type we've seen thus far is that an array variable refers to the *memory address* of the first element in the array. As a result, **it is possible to modify the contents of an array that is passed to a function**. Pass-by-value semantics still apply; the function simply receives a *copy* of the memory address at which the array begins. Here is an example of a function that modifies a C string by overwriting any *trailing* whitespace characters with the null-character. Notice that for C strings we do *not* need to pass the size of the string, since, by convention, the null character marks the end of the string (and thus we can just use the built-in `strlen` function).

```c
#include <stdio.h>
#include <stdlib.h>

void strip_trailing_whitespace(char string[]) {
    int index = strlen(string)-1;
    while (index >= 0) {
        if (isspace(string[index])) {
            string[index] = '\0';
            index -= 1;
        } else {
            // as soon as we encounter first non-whitespace
            // character, get out of loop
            break;
        }
    }
}

int main() {
    char s[128] = "hello\n\n\t";
    strip_trailing_whitespace(s);
    printf("%s\n", s);
    return EXIT_SUCCESS;
}
```

How does this jive with pass-by-value? What happens here is that `s` in `main` holds the memory address of the array, which is allocated on the stack of `main`. When the `strip_trailing_whitespace` function is called, the value of `s` is copied to the parameter `string`, but *the value itself is a memory address*. So the `string` array inside `strip_trailing_whitespace` holds the same memory address as `s` back in `main`. Thus, these two variables *refer to the same array in memory*, as depicted in the figure below. As a result, when we modify the string inside the function, the changes can be observed when we return back to `main`.

No function overloading or default parameters in C

In some languages, e.g., C++, it is permitted to have more than one function definition with the same name, as long as the two definitions differ in the number and data type(s) of parameters. Other languages permit "default" parameters, which means that if a caller chooses *not* to pass a particular argument, the parameter gets a *default* value. Unfortunately, C's syntax does not permit either of these fairly convenient techniques.

7.2.5 A longer example

We'll wrap up this section with one more example. A few things to note:

- The `struct student` has an *embedded* struct field (`struct course_grade`). Actually, an array of `struct course_grade`. One `struct student` would occupy a pretty large chunk of memory. It is left to you

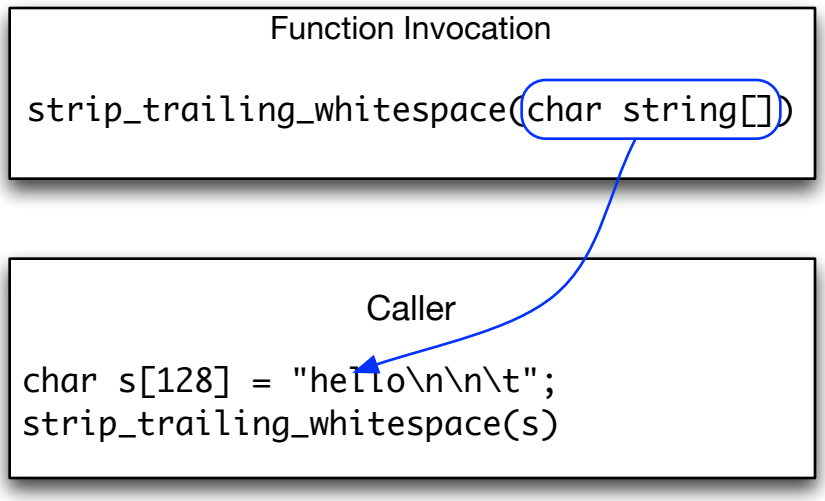


Fig. 1: An array parameter gets a *copy of the memory address* of the array passed into the function, and thus "points" back to the same array contents as can be observed outside the function.

to compute how many bytes, and where any padding is silently inserted by the compiler.

- In `struct student` we need to keep the field `num_courses_completed` to know how many array elements in `courses_completed` are meaningful.
- The `typedef`s on lines 16-17 help to save a few keystrokes with the `struct` usage.
- In `compute_gpa`, we don't need to specify the size of the `grade_t` array, but we do need an additional parameter to tell us how many entries in the array we should consider.
- The initialization syntax for the array of `student_t` just follows the rules we've discussed for array and struct initialization. It is perfectly valid to nest the curly braces where necessary to achieve the correct field initializations.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct course_grade {
5      char course_name[32];
6      char letter_grade;
7  };
8
9  struct student {
10     char name[32];
11     short class_year;
12     int num_courses_completed;
13     struct course_grade courses_completed[48];
14 };
15
16 typedef struct student student_t;
17 typedef struct course_grade grade_t;
18
19 double compute_gpa(grade_t course_list[], int num_courses) {
20     double sum = 0.0;
21     for (int i = 0; i < num_courses; i++) {
22         if (course_list[i].letter_grade == 'A') {
```

(continues on next page)

(continued from previous page)

```
            sum += 4.0;
        } else if (course_list[i].letter_grade == 'B') {
            sum += 3.0;
        } else if (course_list[i].letter_grade == 'C') {
            sum += 2.0;
        } else if (course_list[i].letter_grade == 'D') {
            sum += 1.0;
        }
    }
    return sum / num_courses;
}

void print_student(student_t s, double gpa) {
    printf("%s, Class of %d, GPA: %2.2f\n", s.name, s.class_year, gpa);
}

int main() {
    student_t students[2] = {
        { "A. Student", 2019, 3, { {"Flowerpot construction", 'A'},
                                   {"Underwater basketweaving", 'C'},
                                   {"Dog grooming", 'B'}  },
        },
        { "B. Smart", 2018, 4, { {"Flowerpot construction", 'B'},
                                 {"Underwater basketweaving", 'C'},
                                 {"Cat dentistry", 'C'},
                                 {"Dog grooming", 'C'},  }
        }
    };
    int num_students = sizeof(students) / sizeof(student_t);

    for (int i = 0; i < num_students; i++) {
        double gpa = compute_gpa(students[i].courses_completed, students[i].num_courses_completed);
        print_student(students[i], gpa);
    }
    return EXIT_SUCCESS;
}
```

Compiling and running this code gives the following output:

```
A. Student, Class of 2019, GPA: 3.00
B. Smart, Class of 2018, GPA: 2.25
```

7.3 Storage classes, the stack and the heap

Static variables and the static storage class

The keyword `static` has two, somewhat different, meanings and usages in C.

The first usage is that variables within functions can be prefixed with the `static` keyword to indicate that their value is retained across invocations of the function. For example, consider the following function:

```
void myfunction() {
    static int i = 0;
    i += 1;
    printf("i is now %d\n", i);
}
```

Because of the `static` keyword, the value stored in `i` is retained across multiple calls of `myfunction`. Without the `static` keyword, the output of the function would *always* be `i is now 1`.

The second meaning of `static` in C is to indicate that functions or variables defined outside any function should be *local* to the source file in which they are defined. In *Program structure and compilation* we will discuss header files, compilation, and issues related to this usage of `static` in more detail.

There are two essential "storage classes" in C: *automatic* and *static*. Static storage is somewhat of an advanced concept, and you can refer to the sidebar for a brief discussion.

"Automatic" variables are what we have used exclusively thus far: they are variables that come into existence when a code block is entered, and which are discarded when a code block is exited. This type of allocation and deallocation is done on the call stack of the running program. Recall that all parameters and local variables are allocated on the stack in a last-in, first-out manner. This is exactly the idea behind the "automatic" storage class — memory is *automatically* assigned on the stack[7].

It's worth repeating that all variables in examples we've considered to this point are "automatic" and allocated on the stack. That goes for strings, arrays of various sorts, structures, etc. Most often, we've declared some local variables in `main` (which are allocated on the stack of `main`), and passed parameters into functions (which results in the creation of copies of those parameters in the stack frame of the called function).

Exercises

1. Write a function that takes two `struct fractions`, `a` and `b` and compares them for equality. Return -1 if `a` is less than `b`, 0 if they are equal, and 1 if `a` is greater than `b`.
2. Refactor and modularize the code in exercise 1 in the `struct` chapter. At the very least, write functions to parse a single line into a struct, and to print out a struct.
3. Write a text-based program to play hangperson. Many of you have probably written this sort of program in Python. Test your mettle by writing it in C.

[7] The compiler is responsible for this magic. It must emit the right code so that the stack is managed correctly and variables come into existence and go away at exactly the right point in execution.

CHAPTER

EIGHT

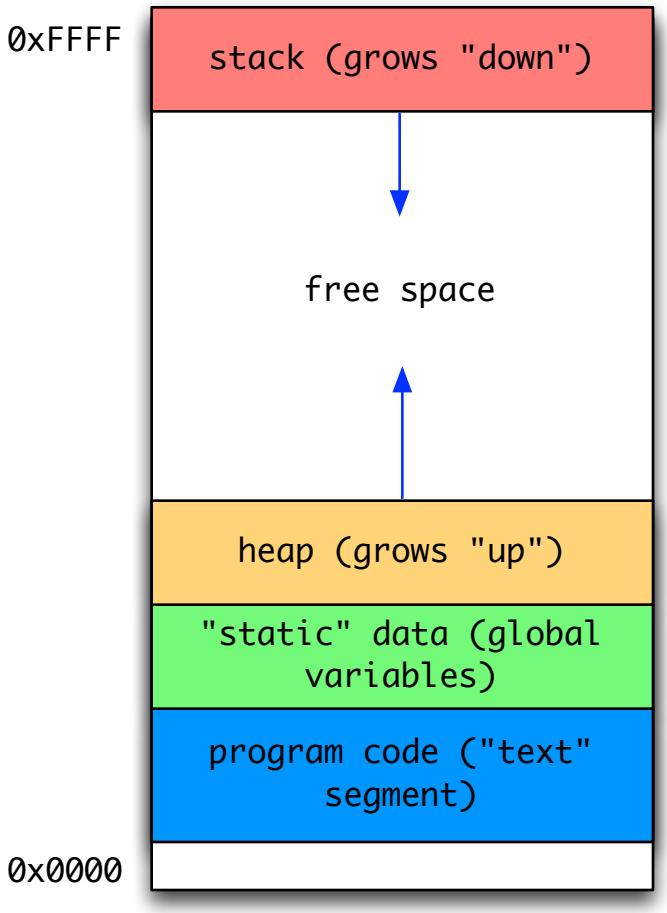
POINTERS AND MORE ARRAYS

The C programming language has a somewhat split personality. On the one hand, it is a *high-level programming language*[1] in that it provides basic control and data abstractions so that a programmer does not have to work in low-level assembly code. On the other hand, it is often considered a fairly *low-level language*[2] due to the fact that it provides only simple data types that directly reflect standard hardware capabilities (e.g., integers, floating point numbers, and simple character strings) and that it allows programmers to directly manipulate memory and memory addresses. In this chapter, we focus on C's capabilities for enabling a programmer to directly manipulate memory and memory addresses.

Process address spaces

As a bit of context for discussing memory addresses and pointers, consider the following depiction of the *address space* of a running program (a "process").

[1] http://en.wikipedia.org/wiki/High-level_programming_language
[2] http://en.wikipedia.org/wiki/Low-level_programming_language



A process in memory typically has (at least) 4 different portions of memory ("segments") dedicated to different purposes. For example, the binary machine code for the program must reside in memory, and a segment is dedicated to storage for global ("static") variables. These portions of memory typically remain *constant* in size, e.g., the amount of memory used for program code does not need to change. There are two segments, however, that are designed to grow and shrink over the lifetime of a process: the *stack* and the *heap*. The stack holds data for each function in progress, including space for local variables, space for parameters, and space for return values. For each function call and return, the size of the stack will grow or shrink, respectively. The heap contains storage for dynamic data structures, e.g., data objects in linked lists, which are managed by the programmer.

8.1 Pointers

A *pointer* is a variable that holds a memory address. The C language allows a programmer to manipulate data *indirectly* through a pointer variable, as well as manipulate the memory address itself stored in the pointer variable.

8.1.1 Declaration syntax

Declaring a new pointer variable is accomplished by using the syntax `<data-type> *<variable-name>;` The asterisk symbol between the data type and variable name indicates that the variable holds a memory

address that refers to a location holding the given data type. For example, the following declaration creates a variable `p` that contains a memory address referring to a location holding an `int` type.

```
int *p; // p points to ???
```

Recall that C does not do any automatic initialization of variables. Thus, the variable `p` will hold an *undefined* memory address after the above declaration. To initialize the pointer so that it points to "nothing", you use `NULL` in C, which is defined as the special address 0.

```
int *p = NULL; // p points to nothing
```

The figure below depicts the state of `p` after this assignment

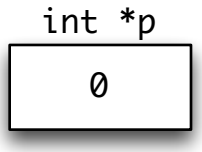


Fig. 1: `p` is `NULL`, or holds the special address 0.

8.1.2 &: Address-of operator

It is often the case that we need to obtain the address of a variable in memory in order to indirectly manipulate its contents through a pointer variable. The address-of operator — `&` — is used for this purpose. For example, the following two lines of code create an integer variable `i` initialized with contents 42, and a *pointer to int* variable `p` which is initialized with *the address of i*. Notice that the `&` goes before the variable for which we want to obtain the address.

```
int i = 42;  // i directly holds the integer 42 (on the stack)
int *p = &i; // p holds the address of i
```

Below is an example depiction of the contents of memory assuming that the variable `i` is stored at (hex) address 0x1004, and `p` is stored in the next four bytes. (Note that this figure assumes 32 bit addressing, since `p` — which holds a memory address — occupies exactly 4 bytes, or 32 bits, in this diagram.)

8.1.3 Dereferencing, or "following" a pointer

Now that `p` "points to" the contents of `i`, we could indirectly modify `i`'s contents through `p`. Essentially what we want to do is to "follow" (or "dereference") the pointer `p` to get to the integer that its address refers to (i.e., `i`), and modify those contents.

The asterisk (`*`) is used as the dereference operator. The basic syntax is: `*` `<pointer-variable>`, which means "obtain the contents of the memory address to which `<pointer-variable>` refers. (Notice that the asterisk goes to the left of the pointer variable that we wish to dereference.) We could use this syntax to increment `i` by one, indirectly through `p`, as follows:

```
int i = 42;  // i directly holds the integer 42
int *p = &i; // p holds address of i
*p = *p + 1; // dereference p (follow pointer), add one to int to which
             // p points, then assign back to int to which p points
printf("%d\n", i); // -> will print 43
```

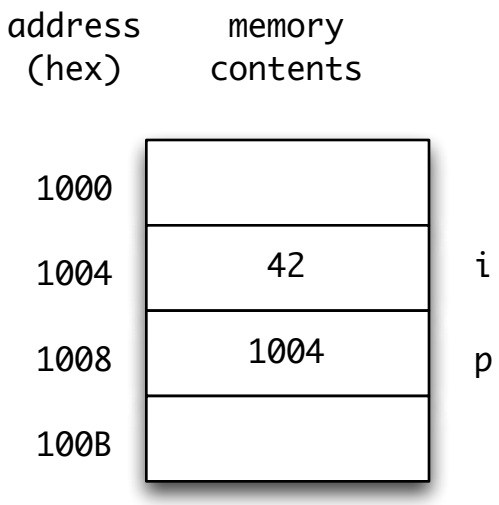


Fig. 2: `i` directly holds the value 42, and `p` holds the address of `i`.

A canonical example for why pointers can be useful is for implementing a function that successfully swaps two values. Here is the code to do it:

```
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int x = 42, y = 13;
    printf("x is %d, y is %d\n", x, y);
    swap(&x, &y);
    printf("x is %d, y is %d\n", x, y);
    return EXIT_SUCCESS;
}
```

The key to this code is that we declare the `swap` function to take two *pointers to ints* as parameters (rather than the two integers themselves). In `main`, we pass *copies of the addresses of x and y*, as shown in the figure above. Inside `swap`, therefore, `a` holds the memory address of `x` (which is back on `main`'s stack) and `b` holds the memory address of `y` (which is also back on `main`'s stack). Through the pointers, we indirectly modify the contents of `x` and `y`.

Uninitialized pointers

When using pointers, there are two entities to keep track of: the pointer itself, and the memory address to which the pointer points, sometimes called the "pointee". There are three things that must be done for a pointer/pointee relationship to work correctly:

1. The pointer must be declared and allocated
2. The pointee must be declared and allocated

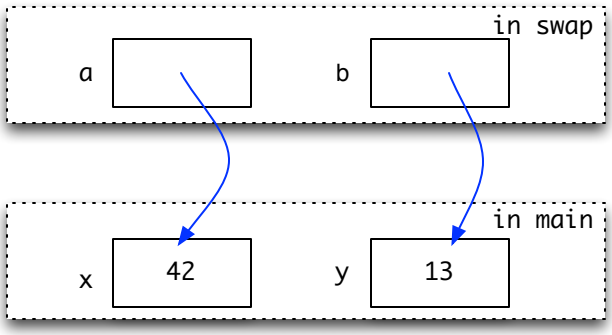


Fig. 3: Inside the `swap` function, `a` holds the address of `x` back on `main`'s stack and `b` holds the address of `y` also on `main`'s stack. With pass-by-value semantics, `a` gets a *copy* of the address of `x` (likewise, `b` gets a *copy* of the address of `y`).

3. The pointer (1) must be initialized so that it points to the pointee (2)

A common error is to do (1), but not (2) or (3). For example:

```
int *p;  // p points to ???
*p = 13; // follow p to some unknown memory location and put 13 there
```

Since C does not do any initialization for the programmer, just declaring a pointer (i.e., step 1) isn't enough for *using* a pointer. In the above code, `p` points to some undefined memory location and the act of writing the integer 13 to that location *may* result in a crash. The crash will likely appear to be *random*, but is entirely due to the fact that `p` was never properly initialized.

To fix this error, `p` must point to some actual `int` in memory, for example:

```
int q = 99;
int *p = &q; // p now is initialized to hold the address of q
*p = 13;
```

It is worth noting that the **`scan-build`** (see *an earlier discussion on scan-build*) static analysis tool can detect problems with uninitialized pointers.

8.1.4 Pointers to `struct`'s

Pointer variables can refer to *any* data type, including `struct` variables. For a `struct`, the syntax for handling pointers can be a bit tricky. To illustrate the trickiness, here is a function that exchanges (swaps) the numerator and denominator of a `struct fraction` (along wht a bit of code to call the function):

```
void flip_fraction(struct fraction *f) {
    int tmp = (*f).denominator;
    (*f).denominator = (*f).numerator;
    (*f).numerator = tmp;
}

struct fraction frac = { 1,2};
flip_fraction(&frac);
```

Why do we need to use parentheses around the `(*f)`? The reason is that the field selection operator (`.`) has higher operator precedence than the dereference operator. Thus, a statement like `*f.numerator` simply does not work: it gets treated by the compiler as `*(f.numerator)`. If `f` is a pointer, `f.numerator` just doesn't

make any sense. As a result, it is necessary to first dereference the struct pointer, *then* access the numerator field.

Because of the awkwardness of requiring the parens for `(*f).numerator` to work right, C provides an operator to access a struct field through a pointer: the *arrow* operator (->):

```
void flip_fraction(struct fraction *f) {
  int tmp = f->denominator;
  f->denominator = f->numerator;
  f->numerator = tmp;
}
```

The above function using the arrow operator has *exactly* the same effect as the more unwieldy version of the `flip_fraction` function above.

8.1.5 Example operating system call with pointers: `gettimeofday`

A standard function for getting the current system time in seconds and microseconds is to use the `gettimeofday` call. This function is declared in the header file <`sys/time.h`> and has the following signature:

```
int gettimeofday(struct timeval *, struct timezone *);
```

where the first argument is a pointer to a `struct timeval`, and the second argument is a pointer to a `struct timezone`. A `struct timeval` has two fields: `tv_sec` and `tv_usec`, which contain the seconds and microseconds after the UNIX epoch (Midnight, January 1, 1970), respectively. This function *fills in* these fields in the `struct timeval` passed to the function (i.e., it modifies the two fields of this struct). `NULL` is normally passed for the timezone argument.

If a programmer wants to get the current system time, a standard way to use this function is to declare a `struct timeval` on the stack of the currently executing function (i.e., as a local variable), then pass the address of this struct to `gettimeofday`, as follows:

```
struct timeval tv;
gettimeofday(&tv, NULL);
// tv.tv_sec and tv.tv_usec now have meaningful values filled in by the gettimeofday function
```

This pattern of passing the address of a stack-allocated struct is fairly common when making various system calls.

The `const` qualifier

The keyword `const` can be added to the left of a variable or parameter type to declare that the code using the variable will not change the variable. As a practical matter, use of `const` is very sporadic in the C programming community. It does have one very handy use, which is to clarify the role of a parameter in a function prototype. For example, in:

```
void foo(const struct fraction* fract);
```

In the `foo()` function prototype, the `const` declares that `foo()` does not intend to change the struct fraction pointee which is passed to it. Since the fraction is passed by pointer, we could not know otherwise if `foo()` intended to change our memory or not. Using `const`, `foo()` makes its intentions clear. Declaring this extra bit of information helps to clarify the role of the function to its implementor and caller.

8.2 Advanced C Arrays and Pointer Arithmetic

8.2.1 Array/pointer duality

Interestingly, C compilers do not meaningfully distinguish between arrays and pointers — a C array variable actually just holds the memory address of the beginning of the array (also referred to as the *base address* of the array). In the following code, we illustrate the *duality* of arrays and pointers by creating 10-element `int` array (`fibarray`) and a *pointer* to an `int` (`fibptr1`). Notice that we directly assign the array variable to an `int *`, which is perfectly legal in C and nicely illustrates the duality between pointers and arrays:

```
int fibarray[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
int *fibptr1 = fibarray;
```

An alternative (and somewhat more explicit) syntax for obtaining the base address of the array is to use the address-of operator with the first element of the array. The following declaration creates yet another pointer variable that refers to the beginning of the array:

```
int *fibptr2 = &fibarray[0]; // get the memory address of the first element of the array
```

Array names are constant pointers

One subtle distinction between an array and a pointer is that the array name where it is declared in the code cannot be modified. In other words, an array name cannot be made to refer to a *different* array or pointer in memory. For example:

```
int ints[100]
int *p;
int i;

ints = NULL;      // NO: cannot change the base address pointer
ints = &i;        // NO
ints = ints + 1;  // NO
ints++;           // NO
```

8.2.2 Pointer arithmetic

The + operator can be used with pointers to access memory locations that reside at some *offset* from a pointer. For example, say that we have the following variable: `int *i`. `i+j` (where j is an integer, *not* a pointer) is interpreted by the compiler as `i + j * sizeof(int)`. Thus, `i+j` yields the memory address of the jth `int` after the address `i` (where we start counting at 0, as you should expect).

A somewhat longer example of adding a pointer and integer together is shown below:

```
int fibarray[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
int *fibptr1 = fibarray;

int a = *(fibptr + 0);  // add 0*sizeof(int) to fibptr address, then dereference (yields the value␣
↪1)
int b = *(fibptr + 2);  // add 2*sizeof(int) to fibptr address, then dereference (yields the value␣
↪2)
```

Again, the syntax `fibptr + 2` is interpreted by the compiler as "get the address of the 2nd integer following the address fibptr".

In fact, array indexing syntax works identically to pointer arithmetic. As a result, square-brace indexing can be used with pointer variables. Moreover, the nice thing about this syntax is that *dereferencing is automatic*. Continuing the code above:

```
int c = fibptr1[5]  // add 5*sizeof(int) to fibptr1 address,
                    // then dereference (automatically!) (yields 8)
```

A totally bizarre implication of the way that C handles array indexing and pointers is that the array name and index value can be inverted!

```
int array[] = { 1, 2, 3};
printf("%d\n", array[1]);   // "normal" indexing
printf("%d\n", 1[array]);   // bizarro inverted indexing, but legal and identical to previous line!
printf("%d\n", *(array+1)); // pointer arithmetic syntax
printf("%d\n", *(1+array)); // pointer arithmetic syntax, with operands reversed
```

The above code is purely an illustration — don't write code with inverted indexing! Although it is legal, it is a "feature" that makes the code harder to read since nobody writes indexes like that.

8.3 Dynamic memory allocation

We started this chapter by outlining how memory is organized within a single running program, or process (see *Process address spaces*, above). So far, we have just used local and parameter variables, which result in *stack-allocated* memory. In this section, we discuss how to *dynamically* allocate and deallocate blocks of memory on the heap. C requires that a program *manually* manage heap-allocated memory through explicit allocation and deallocation. In contrast, a language like Java only requires that a programmer explicitly allocate memory, but the language runtime handles automatic deallocation through a process called *garbage collection*.

8.3.1 `malloc` and `free`

The built-in functions `malloc` and `free` are used to manually allocate and deallocate blocks of heap memory. These functions are declared in the header file <`stdlib.h`> (i.e., you must `#include` this file) and work as follows:

Pointing into the `void`

Notice that the `malloc` function returns a "pointer to void" (`void *`), and `free` takes a `void *` as a parameter. By convention in C, a pointer which does not point to any particular type is declared as `void*`. Sometimes `void*` is used to force two bodies of code not to depend on each other, since `void*` translates roughly to "this points to something, but I'm not telling you (the client) the type of the pointee exactly because you do not really need to know." That's exactly the case with `malloc` and `free`: the `malloc` function cannot possibly know what the caller wants the new memory block allocated on the heap to contain, and neither can the `free` function know what data type some memory block points to.

Note that a `void *` cannot be dereferenced — the compiler prevents this. The pointer must be cast to a pointer to some concrete type in order to be dereferenced.

Also, interestingly, `NULL` is usually defined as `(void*)0`.

`void* malloc(size_t size)` `malloc` takes one parameter: the number of bytes to allocate on the heap. It returns a "generic pointer" (i.e., `void *`) that refers to a newly allocated block of memory on the heap.

If there is not enough memory on the heap to satisfy the request, `malloc` returns `NULL`.

`void free(void* block)` The mirror image of `malloc`, `free` takes a pointer to a heap block previously returned by a call to `malloc` and returns it to the heap for re-use. After calling `free`, the caller should not access any part of the memory block that has been returned to the heap.

Note that all of a program's memory is deallocated automatically when it exits, so a program *technically* only needs to use `free` during execution if it is important for the program to recycle its memory while it runs — typically because it uses a lot of memory or because it runs for a long time. However, **it is always good practice to free what ever you malloc**. You should not rely on the fact that a program does not run long or that you *think* it does not use a lot of memory.

Here is some example code that uses `malloc` and `free` to allocate a block of `struct fraction` records (basically an array, but not declared as an array), fill each one in with user input, invert each one, then print them all out. Notice that each of the functions `get_fractions`, `invert_fractions`, and `print_fractions` accesses each `struct fraction` in different ways: by index, and by pointer arithmetic. Note specifically that the `invert_fractions` function modifies the `fracblock` pointer (by "incrementing it by 1, which makes the pointer advance to the next `struct fraction`), but since that function just gets a *copy* of the pointer to the `struct fraction` this is totally ok.

```
#include <stdio.h>
#include <stdlib.h> // for malloc and free

struct fraction {
    int numerator;
    int denominator;
};

void get_fractions(struct fraction *fracblock, int numfrac) {
    char buffer[32];
    for (int i = 0; i < numfrac; i++) {
        printf("Enter numerator for fraction %d: ", i+1);
        fgets(buffer, 32, stdin);
        int numerator = atoi(buffer);
        printf("Enter denominator for fraction %d: ", i+1);
        fgets(buffer, 32, stdin);
        int denominator = atoi(buffer);

        // use array syntax to fill in numer/denom for the ith fraction
        fracblock[i].numerator = numerator;
        fracblock[i].denominator = denominator;
    }
}

void invert_fractions(struct fraction *fracblock, int numfrac) {
    for (int i = 0; i < numfrac; i++) {
        int tmp = fracblock->numerator;
        fracblock->numerator = fracblock->denominator;
        fracblock->denominator = tmp;

        fracblock += 1; // pointer arithmetic:
                        // advance the pointer by 1 struct fraction
    }
}

void print_fractions(struct fraction *fracblock, int numfrac) {
    for (int i = 0; i < numfrac; i++) {
        // use pointer-arithmetic syntax to get numerator/denominator
```

(continues on next page)

(continued from previous page)

```
        // for each fraction

        printf("%d: %d/%d\n", i+1, (fracblock+i)->numerator,
                                   (fracblock+i)->denominator);
    }
}

int main() {
    char buffer[32];
    printf("How many fractions to make? ");
    fgets(buffer, 32, stdin);
    int numfrac = atoi(buffer);

    // allocate a block of numfrac fractions from the heap
    struct fraction *fractions = malloc(sizeof(struct fraction) * numfrac);

    // call function to "fill-in" each fraction
    get_fractions(fractions, numfrac);
    invert_fractions(fractions, numfrac);
    print_fractions(fractions, numfrac);

    free(fractions); // return block of fraction memory to the heap

    return EXIT_SUCCESS;
}
```

8.3.2 Memory leaks and dangling pointers

Note that in the above example code, we have exactly 1 call to `malloc` and exactly 1 matching call to `free`. If you do not have a matching `free` call for each malloc, your program has a *memory leak*. Memory leaks are especially problematic for long-running programs (e.g., web browsers are often implicated in memory leak problems[3]). The following program is one example of a pretty horrible leak: there is a `malloc` call in a loop, but no matching `free`. Even worse, we completely lose the ability to access the memory block in the previous iteration of the loop by re-assigning to `memory_block` each time through the loop. Note also that assigning `NULL` doesn't free a block; it simply makes a block inaccessible to the program.

```
for (int i = 0; i < BIGNUMBER; i++) {
    char *memory_block = malloc(1024*1024);  // allocate a chunk on the heap
    // do nothing else!
}
memory_block = NULL;  // doesn't free anything!  we just lost our access
                      // to the memory block most recently allocated, so
                      // we've created a hopeless memory leak!
```

A *dangling pointer* is a pointer that refers to a invalid block of memory, either to an undefined memory address or to a memory block that has already been freeed, and should thus be considered inaccessible. For example:

```
int *p = malloc(sizeof(int));
*p = 42;
int *q = p;  // q is a pointer; now it just holds the same address as p
```

(continues on next page)

[3] Just search for "Firefox memory leak" and you'll find plenty of posts not unlike the following: https://support.mozilla.org/en-US/questions/1006397

(continued from previous page)

```
printf("q is %d\n", *q); // 42
printf("p is %d\n", *p); // 42
free(p); // free p.
printf("p is %d\n", *p); // NO!  p is invalid because we just free'd it!
printf("q is %d\n", *q); // Double NO!  since we free'd p, q is a "dangling pointer"
                         // since it pointed to the same memory block!
```

scan-build and valgrind

Valgrind is a pretty excellent tool for helping to ferret out memory leaks, memory trashing, and any other type of memory corruption error that can happen in C programs. To run a program with valgrind, you can just type **valgrind <program>**.

There are *many* command-line options to change the behavior or output of valgrind. Type **valgrind -h** for help (or **man valgrind**). See http://valgrind.org for more information on this great tool.

Besides **valgrind**, the **scan-build** tool is also incredibly helpful. It is also usually faster and with better (easier to understand) output. See a *previous description of scan-build* as well as https://clang-analyzer.llvm.org/scan-build.html for more information.

8.3.3 Advantages and disadvantages of heap-allocated memory

Heap-allocated memory makes it possible to create linked lists, dynamically-sized arrays and strings, and more exotic data structures such as trees, heaps, and hashtables. Manually allocating and deallocating memory can be a pain, though. As a result, you probably want to be strategic about whether to use stack-allocated memory (e.g., local arrays and variables) or heap-allocated memory in a program. Here are some key advantages and disadvantages to help you consider what is right for a given situation:

Advantages to heap allocation

- The size of an array, string, or some other data structure can be defined at run time. With stack-allocated arrays, for example, you typically need to specify a "reasonable upper bound" for the size of the array, and somehow deal with the consequences if the size of the array is exceeded.
- A block of memory will exist until it is explicitly deallocated with a call to `free`. For stack-allocated memory, the memory is automatically deallocated when a function is exited, which is totally inappropriate for data structures such as linked lists.
- You can dynamically *change* the size of the array, string, or some other data structure at run time. There is a built-in `realloc` function that can help with this (see **man realloc**),

Disadvantages to heap allocation

- You have to remember to allocate and deallocate a data structure, and you have to get it right. This is harder than it sounds, and when things go wrong the program will either exhibit unexpected (buggy) behavior, or crash in a ball of flames. Debugging can be hard.
- You have to remember to deallocate a memory block exactly once when you are done with it, and you have to get that right. Also, harder than it looks. For example, calling `free` on the same memory block *twice* is an error, and typically causes a crash.

8.3.4 Dynamic Arrays

Since arrays are just contiguous areas of bytes, you can allocate your own arrays in the heap using `malloc`. It is also fairly straightforward to resize an array as necessary (i.e., to grow it to accommodate more data items). The following code allocates two arrays of 1000 ints — one in the stack the usual "local" way, and one in the heap using `malloc`. Other than the different allocations, the two are syntactically similar in use.

```
int a[1000]; // allocate 1000 ints in the stack
int *b = malloc(sizeof(int) * 1000);  // allocate 1000 ints on the heap
a[123] = 13;      // just use good ol' [] to access elements
b[123] = 13;      // in both arrays
free(b);          // must call free on the heap-allocated array
```

To grow the heap-allocated array, we could do something like the following. (Note that the following code uses `memcpy`, which accepts three parameters: a destination address, a source address, and the number of bytes to copy):

```
int *arr = malloc(sizeof(int) * 1000); // 1000 ints on the heap

// assume we need to grow the array

int *newarr = malloc(sizeof(int) * 2000); // double your integer pleasure
memcpy(newarr, arr, 1000*sizeof(int)); // copy over contents of old array
free(arr);    // free old array
arr = newarr; // arr now points to new, larger block
```

8.3.5 C strings revisited

Although we have used arrays of `char` to hold C strings thus far, a much more common way to declare the type of a C string is `char *`. This shouldn't be particularly surprising, since arrays and pointers are treated nearly synonymously in C. That's not to say that stack-allocated C strings as arrays aren't useful. Indeed, they are very commonly used. It is, however, often necessary to copy and manipulate strings in memory, and using stack or statically allocated arrays becomes quite difficult.

As an example, say that we need to "escape" an HTML string to replace any occurrence of < with `<` (lt: "less-than") and any occurrence of > with `>` (gt: "greater-than"). (There are other characters that are replaced when "properly" escaping an HTML string; we're just focusing on these two characters in this example.) Since the string will "grow" as we escape it, dynamic memory allocation has obvious benefits. Here is the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int count_escapees(const char *htmltext) {
    int count = 0;
    for (int i = 0; i < strlen(htmltext); i++) {
        if (htmltext[i] == '<' || htmltext[i] == '>') {
            count += 1;
        }
    }
    return count;
}

void doescape(const char *htmltext, char *expandedtext) {
```

(continues on next page)

(continued from previous page)

```
    int j = 0;
    for (int i = 0; i < strlen(htmltext); i++) {
        if (htmltext[i] == '<') {
            strcpy(&expandedtext[j], "&lt;");
            j += 4;
        } else if (htmltext[i] == '>') {
            strcpy(&expandedtext[j], "&gt;");
            j += 4;
        } else {
            expandedtext[j] = htmltext[i];
            j += 1;
        }
    }
}

char *escapehtml(const char *htmltext) {
    int count = count_escapees(htmltext);
    int origlen = strlen(htmltext);
    int expandedlen = origlen + count * 4 + 1;
    char *expandedtext = malloc(sizeof(char) * expandedlen);
    doescape(htmltext, expandedtext);
    return expandedtext;
}

int main() {
    const char *orig = "<a href=\"badurl\">a link!</a>";
    char *escaped = escapehtml(orig);
    printf("Original: %s\n", orig);
    printf("Escaped: %s\n", escaped);
    free(escaped);
    return EXIT_SUCCESS;
}
```

8.3.6 Linked lists

One of the most commonly used dynamic data structures is the *linked list*. A standard definition of a linked list node in C, in which each node contains an integer, is as follows:

```
struct node {
    int value;
    struct node* next;
};
```

Notice that there's something of a circular definition and usage here (i.e., inside the definition of `struct node`, we declare a `struct node` as a field). C is perfectly happy with that circularity.

Manipulating nodes in a linked list generally involves allocating new nodes on the heap, linking in new nodes to the list, and/or modifying node pointers in other ways. Here is a bit of code for adding a new node to a list by inserting in the front:

```
struct node *insert(struct node *head, int new_value) {

    struct node *new_node = malloc(sizeof(struct node));
    new_node->value = new_value;
    new_node->next = head;  // next ptr of new node refers to head of old list
```

(continues on next page)

(continued from previous page)

```
    return new_node;
}
```

A function to traverse a list and print each value out might look like the following. Notice that since the `print_list` function gets a *copy* of the head of the list, it is safe to modify that pointer within the `print_list` function.

```
void print_list(struct node *head) {
    int i = 0;
    while (head != NULL) {
        printf ("Node %d has the value %d\n", i+1, head->value);
        head = head->next;  // advance the list pointer
        i += 1;
    }
}
```

8.3.7 Pointers to pointers, etc.

Some functions in the C standard library take pointers-to-pointers ("double pointers"), and you will likely encounter situations in which it is *useful* to make pointers-to-pointers. One example of such a situation occurs when a function needs to allocate and initialize heap memory for a caller. Here is an example in code:

```
int copy_string(char **dest, const char *source) {
    int buffer_size = strlen(source) + 1;
    *dest = malloc(sizeof(char) * buffer_size);
    if (!*dest) {
        return -1;  // failure!
    }
    strlcpy(*dest, source, buffer_size);
    return 0; // success
}

char *ptr;
copy_string(&ptr, "here's a string!");
printf("%s\n", ptr);
// don't forget: need to eventually free(ptr)!
```

In the above code, the `copy_string` function takes a "pointer to a pointer to a char" as the first parameter, and a constant C string as the second parameter. The function allocates a new block of memory using `malloc` and copies the `source` string into a `dest` string.

Why can't the first parameter be `char *`? The reason has to do with pass-by-value function parameters: since we want to *modify* what `dest` points to, we need to access that pointer indirectly or the modification only happens to a local variable. (This is exactly the same situation we encountered with the failed `swap` function.) Since we can indirectly access the `dest` pointer (i.e., via the pointer to the pointer), the change we make is observable to the caller of the function.

Outside the function, we declare a normal `char *` string variable (`ptr`), then pass *the address of ptr* to the function, which creates the pointer-to-a-pointer. When we return from the function, `ptr` has been modified (specifically, the address held in the variable `ptr` has been modified). As a side-effect of the way this function is implemented (i.e., it uses `malloc`), we must remember to eventually call `free` on `ptr` in order to avoid a memory leak.

Exercises

1. Consider the following code. Identify exactly what is allocated on the stack and what is allocated on the heap:

```
int main() {
    int *p = NULL;
    p = malloc(sizeof(int)*42);
    char arr[1000];
    char *p = &arr[10];
    char *q = malloc(64);
    char *r = &q[10];
}
```

2. Write a function that mimics the built-in `strdup` function.
3. Write a `clear_list` function that takes a pointer to a linked list (`struct node *`) and calls free for each element in the list (i.e., to completely deallocate the list).
4. Create a `clone_list` function that takes a pointer to a linked list (`struct node *`) and returns a completely cloned copy of the list (i.e., there are exactly the same number of nodes in the new list, with the exactly the same values in the same order, but the old list is left totally unmodified).
5. Create a `reverse_list` function that accepts a pointer to a `struct node` and returns a pointer to the list with all elements now in reversed order.
6. Write a function that appends a new value to the end of a linked list. The function should return a pointer to the head of the list.
7. Write new implementations of the various linked list functions we've seen and written so far but instead of having an int as the value, use a `char *` (dynamically allocated C string).
8. Write a function that accepts the name of a text file, and allocates and returns a C string that contains the *entire* contents of the file. (If you're familiar with the basic file I/O API of Python, this function should work like the `read()` method of a file object.)
9. Rewrite the `escapehtml` function so that it accepts a *pointer to a pointer to a char* (`char **`), allocates a new string that contains the escaped string and *assigns* the newly allocated string to the C string pointer to by the pointer argument to the function. For example, if the parameter is `char **str`, the variable `*str` refers to the C string containing the HTML text to be escaped. When the function concludes, `*str` should *now* refer to the C string containing the escaped HTML text. You should be sure to `free` the original unescaped C string.
10. Rewrite the `escapehtml` function so that it accepts a *pointer to a char* (not a *const* pointer), and modifies the string *in place* to escape each < and >. You should use the built-in C library function `realloc` to dynamically reallocate heap memory allocated to the string passed into the function. You'll need to read the `man` page for `realloc` to understand how this function works.
11. Create a couple C structs to implement a dynamically growable/shrinkable stack data structure (a stack of integers) like the following:

```
struct intnode {
  int value;
  struct intnode *next;
};

typedef struct {
  struct intnode *top;
} stack_t;
```

Write five functions to create, destroy/deallocate, and perform standard operations on a stack with these definitions. Notice that the `stack_t` structure just contains a pointer to the top of the stack, and that stack is just implemented as a linked list of `struct intnode`. For the `pop` function, you can assume in your implementation that the stack is non-empty (i.e., it would be an error on the part of the user of the stack if `pop` was called on an empty stack).

- `stack_t* allocate_stack(void);`
- `void deallocate_stack(stack_t *);`
- `int empty(stack_t *);`
- `int pop(stack_t *);`
- `void push(stack_t *, int);`

CHAPTER

NINE

PROGRAM STRUCTURE AND COMPILATION

9.1 The compilation process

For a C program of any reasonable size, it is convenient to separate the functions that comprise the program into different text files. There are standard ways to organize source code in order to allow the functions in separate files to cooperate, and yet allow the compiler to *build* a single executable.

The process of compiling C programs is different than with Java, and has important implications for how source code must be organized within files. In particular, **C compilers make a single (top-to-bottom) pass over source code files.** This process is *very much unlike the Java compiler*, which may make multiple passes over the same file, and which may automatically compile *multiple* files in order to resolve code dependencies (e.g., if a class is used in one file but defined in another, the compiler will compile *both* files). In C, it is entirely up to the programmer to decide which files need to be compiled and linked to produce an executable program.

There are three basic steps involved in compiling a C program: *preprocessing*, *compilation* of C source code to machine code (or assembly) (also called *object code*), and *linking* of multiple object files into a single binary executable program. Each of these steps are described below.

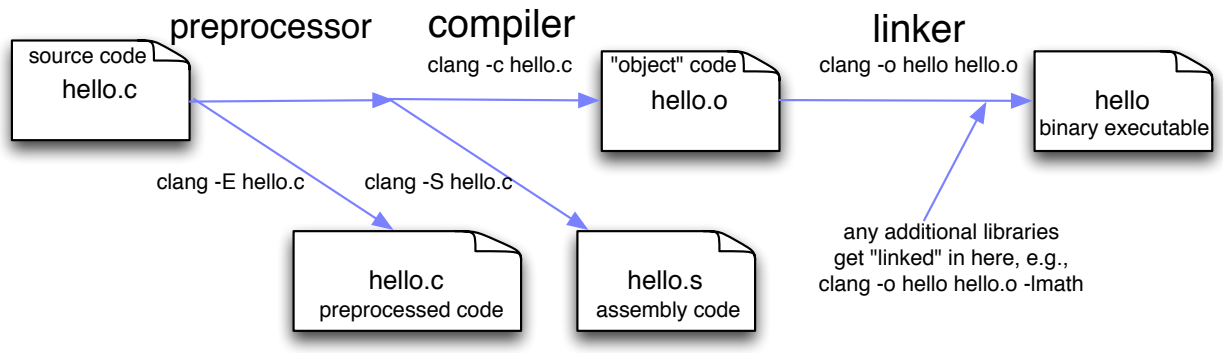


Fig. 1: The three basic steps when compiling C programs are *preprocessing*, *compilation*, and *linking*.

9.1.1 The preprocessing step

The preprocessing step happens just prior to the compilation phase. The C preprocessor looks for any *preprocessor directives* in the source code, which are any lines starting with #. The preprocessor then performs some actions specified by the directive. The text resulting from the preprocessor's action is then fed directly (and automatically) to the compilation phase.

Since a C compiler makes a single pass over a `.c` file, it must be made aware of all the types and signatures in order to correctly and successfully complete the compilation process. That is, if an unknown data type is encountered in the single top-to-bottom pass, the compiler will report an error. For example, here is some source code that will *not* compile correctly:

```
#include <stdlib.h>

int main() {
    struct function f1 = { 1,2};
    return EXIT_SUCCESS;
}

struct function {
    int numerator;
    int denominator;
};
```

Why does it fail? Simply because the definition of `struct function` comes *after* its first use. To make the code correctly compile, the `struct` definition must precede `main`.

Header (`.h`) and source (`.c`) files

Because of the single-pass top-to-bottom operation of C compilers, each source file (each `.c` file) must identify all data types and function signatures that are used in that file in order to make the code successfully compile. The standard practice in C is to define any types and declare any functions in **header** files (`.h` files) in order to facilitate the compilation process. In one sense, you can think of the `.h` files as containing the "external interfaces" (i.e., the API) and data types used for a set of functions, and the corresponding `.c` file as containing the actual function definitions.

For example, say that we want to define the `struct fraction` type and a couple utility functions that can be used in other `.c` files. We might define a `fraction.h` file that contains the following:

```
struct fraction {
    int numerator;
    int denominator;
};

void print_fraction(const struct fraction *);
void invert_fraction(struct fraction *);
```

Notice that this header file contains the `struct` definition, and two **function prototypes**. A "prototype" for a function gives its name and arguments but not its body. The function parameters do not even have to have variable names (as they're shown above), but there's no problem if they *do* include the parameter names.

The corresponding `fraction.c` file might contain the following:

```
#include "fraction.h"

void print_fraction(const struct fraction *f) {
    printf("Fraction: %d/%d\n", f->numerator, f->denominator);
}

void invert_fraction(struct fraction *f) {
    int tmp = f->numerator;
    f->numerator = f->denominator;
    f->denominator = tmp;
}
```

Notice that the first line of code in `fraction.c` is `#include "fraction.h"`. Any line of code that begins # is called a **preprocessor directive**. We have used `#include` quite a bit so far. Its meaning is simply to *directly replace* the `#include` directive with the text in the specified file name.

A file that *uses* the fraction utility functions in a file called `test.c` might look like the following:

```
#include "fraction.h"  // include struct fraction definition and
                       // fraction utility function prototypes,
                       // as well as other headers like stdlib.h

int main() {
    struct fraction f = {2,3};
    invert_fraction(&f);
    print_fraction(&f);
    return EXIT_SUCCESS;
}
```

Preprocessor directives

There are several preprocessor directives that can be listed in C source code. `#include` and `#define` are the two most common, but there are others.

#include

As we've already seen, the `#include` directive reads in text from different files during the preprocessing step. `#include` is a very unintelligent directive — the action is simply to paste in the text from the given file. The file name given to `#include` may be included in angle brackets or quotes. The difference is that *system* files should be enclosed in angle brackets and any *user* files should be enclosed in quotes.

#define

The `#define` directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, `#define` is extremely unintelligent — it just does textual replacement without any code evaluation. `#define` statements are used as a crude way of establishing symbolic constants or *macros*. Generally speaking, you should prefer to use `const` values over `#define` directives.

Here are examples of quasi-constant definitions:

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

Later code can use the symbols `MAX` or `SEVEN_WORDS` which will be replaced by the text to the right of each symbol in its `#define`.

Simplistic *macro* functions can also be defined with `#define` directives. For example, a commonly used macro is `MAX`, which takes two parameters and can be used to determine the larger of two values:

```
#define MAX(a,b) (a > b ? a : b)
```

Again, the `#define` directive is incredibly unintelligent: it is simply smart enough to do textual replacement. For example, the following code:

```
int a = MAX(c, d);
```

would be replaced by the preprocessor with the following:

```
int a = (c > d ? c : d);
```

While `MAX` is often referred to as a *macro function* (or simply macro), it does not operate as a function at all. The programmer can (somewhat) treat the macro as a function, but the effect is just an illusion created by the C preprocessor.

#if

At the preprocessing phase, the symbolic names (and values) defined by `#define` statements and predefined by the compiler can be tested and evaluated using `#if` directives. The `#if` test can be used at the preprocessing phase to determine whether code is included or excluded in what is passed on to the compilation phase. The following example depends on the value of the `FOO` `#define` symbol. If it is true (i.e., non-zero), then the `"aaa"` lines (whatever they are) are compiled, and the `"bbb"` lines are ignored. If `FOO` is false (i.e., 0), then the reverse is true.

```
#define FOO 1

...

#if FOO
    aaa
    aaa
#else
    bbb
    bbb
#endif
```

Interestingly (and usefully), you can use `#if 0 ...#endif` to effectively comment out areas of code you don't want to compile, but which you want to keep in the source file.

Multiple #includes

It is invalid in C to declare the same variable or `struct` twice. This can easily happen if a header file is `#included` twice. For example, if a source code file includes header file A and B, and header file B *also* includes header file A, the contents of header file A will be included *twice*, which may cause problems.

A standard practice to avoid this problem is to use the `#ifndef` directive, which means "if the following symbol is not defined, do the following". The `#define` symbol is often based on the header file name (as in the following), and this practice

This largely solves multiple `#include` problems.

```
#ifndef __FOO_H__
#define __FOO_H__  // we only get here if the symbol __FOO_H__ has not been previously defined

<rest of foo.h ...>

#endif // __FOO_H__
```

`static` functions

There is yet another meaning to the keyword `static` in the context of global variables and functions. Specifically:

1. A function may be declared `static`, in which case it can only be used in the same file, below the point of its declaration. The meaning of `static` in this case is essentially that the function is "private" to the file. That is, it can only be used by other functions within the same file, but not from within another `.c` file.
2. The `static` keyword can also be used with global variables in a `.c` file (i.e., variables defined outside any function). The meaning in this case is the same with `static` functions: the variable is "private" to the `.c` file and cannot be accessed or used from other `.c` files.

For example, here are definitions of a static (private) variable and static (private) function within a `.c` source file:

```
// this variable is not "visible" to any functions in some other .c file
static int private_counter = 0;

// this function is not "visible" to any functions in some other .c file
static void add_to_counter(int increment) {
    // ok to use the private/static variable from this function,
    // since it is in the same file
    private_counter += increment;
}
```

Invoking the preprocessor

Normally, you do not need to do anything special to invoke the preprocessing phase when compiling a program. It is, however, possible to *only* invoke the preprocessing phase (i.e., no compilation or anything else), and also to define new preprocessor symbols on the command line.

To invoke just the preprocessor in **clang**, you can use the command **`clang -E sourcefile.c`**. **clang** has another command line option to just run the preprocessor and check code syntax: **`clang -fsyntax-only sourcefile.c`**.

To define new preprocessor symbols (i.e., just like `#define`), the -D option can be used with **clang**, as in **`clang -DSYMBOL`**, or **`clang -DSYMBOL=VALUE`**.

9.1.2 The compilation step

The compilation step takes as input the result from the preprocessing stage. Thus, any # directives have been processed and are removed in the source code seen by the compiler.

The compilation stage can produce either assembly code or an *object file* as output. Typically, the object code is all that is desired; it contains the binary machine code that is generated from compiling the C source. There are a few different relevant compiler options at this stage:

`clang -S sourcefile.c` Produces assembly code in sourcefile.s

`clang -c sourcefile.c` Produce object file (binary machine code) in sourcefile. This is the more common option to employ for the compilation stage. When all source files have been compiled to object code (`.o` files), all the `.o` files can be *linked* to produce a binary executable program.

Some additional compiler options that are useful at this stage:

option	meaning
-g	include information to facilitate debugging using a program like **gdb**.
-Wall	Warn about any potentially problematic constructs in the code.

9.1.3 The linking phase

The linking stage takes 1 or more object files and produces a binary executable program (i.e., a program that can be directly executed on the processor). It requires two things: that the implementations for any functions referenced in any part of the code have been defined, and that there is exactly one `main` function defined.

Options for linking

In the simplest case, there is only one source file to preprocess, compile, and link. In that case, the same command line we've used with **`clang`** so far does the trick:

```
clang -g -Wall inputfile.c -o myprogram
```

or, if you've already compiled inputfile.c to inputfile.o, just:

```
clang -g -Wall inputfile.o -o myprogram
```

In a more "interesting" case, there is more than one file to compile and link together. For each source file, you must compile it to object code. Following that, you can link all the object files together to produce the executable:

```
clang -g -Wall file1.c -c
clang -g -Wall file2.c -c
clang -g -Wall file3.c -c
clang -g file1.o file2.o file3.o -o myprogram
```

If you use functions from the standard C library, you don't need to do anything special to link in the code that implements the functions in that library. If, however, your program uses a function from an *external* library like the `math` library (see **`man 3 math`**; it contains functions such as `log2`, `sqrt`, `fmod`, `ceil`, and `floor`), the library to be linked with must be specified on the command line. The basic command is:

```
clang -g -Wall inputfile.o -o outputfile -lmath
```

The `-l` option indicates that some external library must be linked to the program, in this case the `math` library.

The main function

The execution of a C program begins with the function named `main`. All of the files and libraries for the C program are compiled together to build a single program file. That file must contain exactly one `main` function which the operating system uses as the starting point for the program. `main` returns an int which, by convention, is 0 if the program completed successfully and non-zero if the program exited due to some error condition. This is just a convention which makes sense in shell oriented environments such as UNIX.

Command-line arguments to a program

For many C programs, it is useful to be able to pass various command-line arguments to the program through the shell. For example, if we had a program named `myprogram` and we wanted to give it the names of several text files for it to process, we might use the following command line:

```
$ ./myprogram file1.txt file2.txt file3.txt
```

Each of the file names (file1-3.txt) is a command-line parameter to the program, and can be collected through two parameters to `main` which are classically called `argc` and `argv` and are declared as follows:

```
int main(int argc, char *argv[]) {
    // ...
}
```

The meaning of these parameters is:

argc The number of command-line arguments given to the program, *including* the program name

argv An array of C strings which refer to each of the command-line parameters. Note that `argv[0]` is *always* the name of the program itself. For example, in the above command line, `argv[0]` would be `"./myprogram"`.

A simple program that traverses the array of command-line arguments and prints each one out could be written as follows:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("argument %d is %s\n", i, argv[i]);
    }
    return EXIT_SUCCESS;
}
```

There is a C library function called `getopt` that enables parsing of options in more flexible ways. See **`man 3 getopt`** for more information.

9.2 Invariant testing and `assert`

Array out of bounds references are an extremely common form of C run-time error. You can use the `assert()` function to sprinkle your code with your own bounds checks. A few seconds putting in `assert` statements can save you hours of debugging.

Getting out all the bugs is the hardest and scariest part of writing a large piece of software. Adding `assert` statements are one of the easiest and most effective helpers for that difficult phase.

```
#include <assert.h>
#define MAX_INTS 100

void somefunction() {
    // ...

    int ints[MAX_INTS];
    i = foo(<something complicated>);
    // i should be in bounds,
    // but is it really?
    assert(i>=0);           // safety assertions
    assert(i<MAX_INTS);
    ints[i] = 0;

    // ...
```

Depending on the options specified at compile time, the `assert()` expressions will be left in the code for testing, or may be ignored. For that reason, it is important to only put expressions in `assert()` tests which do not need to be evaluated for the proper functioning of the program.

```
int errCode = foo();      // yes --- ok
assert(errCode == 0);
if (assertfoo() == 0) ... // NO, foo() will not be called if
                          // the compiler removes the assert()
```

CHAPTER

TEN

C STANDARD LIBRARY FUNCTIONS

10.1 Precedence and Associativity

Order	Operation	Associativity
1.	function-call(), `[] -> .`	L to R
2.	`! ~ ++ -- + - *``(ptr deref) ``sizeof &` (addr of) (all unary operations are the same precedence)	*R to L*
3.	`* / %` (the top tier arithmetic binary ops)	L to R
4.	`+ -` (second tier arithmetic binary ops)	L to R
5.	`< <= > >= == !=`	L to R
6.	in order: `& ^ | && ||` (bitwise before Boolean)	L to R
7.	= and all its variants	*R to L*
8.	`,` (comma)	L to R

One common pitfall with the above precedence fields is that:

```
*structptr.field
```

never works. The field selection operator (`.`) has higher precedence than the dereference operator (*), thus the expression is evaluated as `*(structptr.field)` instead of the (usually) desired `(*structptr).field`. A way to avoid this problem altogether is to write `structptr->field`.

10.2 Standard Library Functions

Many basic housekeeping funcions are available to a C program in form of standard library functions. To call these, a program must `#include` the appropriate header file. All modern compilers link in the standard

library code by default, so all that is needed is to include the correct header file. The functions listed in the next most commonly used ones, but there are many more which are not listed here.

header file	types of functions available
stdio.h	file input and output, e.g., `printf`
ctype.h	character tests, e.g., `isspace`
string.h	string operations
stdlib.h	utility functions, e.g., `atoi`, `rand`, `abs`
math.h	mathematical functions, e.g., `sin`, `pow`
assert.h	the `assert` debugging macro
stdarg.h	support to create functions that take a variable number of parameters
signal.h	support for exceptional condition signals
time.h	date and time, e.g., `time`
sys/time.h	other date/time functions, e.g., `gettimeofday`
limits.h	constants which define type range values such
float.h	as `INT_MAX`, `FLOAT_MAX`
stdbool.h	`bool` type (requires `-std=c99` compiler flag)

10.3 stdio.h

`stdio.h` is a very common file to include. It includes functions to print and read strings from files and to open and close files in the file system.

`FILE* fopen(const char* fname, const char* mode);` Open a file named in the filesystem and return a FILE* for it. Mode = "r" read, "w" write, "a" append, returns NULL on error. The standard files stdout, stdin, stderr are automatically opened and closed for you by the system.

`int fclose(FILE* file);` Close a previously opened file. Returns EOF on error. The operating system closes all of a program's files when it exits, but it's tidy to do it beforehand. Also, there is typically a limit to the number of files which a program may have open simultaneously.

`int fgetc(FILE* in);` Read and return the next unsigned char out of a file, or EOF if the file has been exhausted. (detail) This and other file functions return ints instead of a chars because the EOF constant they potentially is not a char, but is an int. getc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`char* fgets(char* dest, int n, FILE* in)` Reads the next line of text into a string supplied by the caller. Reads at most n-1 characters from the file, stopping at the first 'n' character. In any case, the string is '0' terminated. The 'n' is included in the string. Returns NULL on EOF or error. There's also a `gets` function, but *you should never use it!* (read the `man` page for why).

`int fputc(int ch, FILE* out);` Write the char to the file as an unsigned char. Returns ch, or EOF on err. putc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`int ungetc(int ch, FILE* in);` Push the most recent fgetc() char back onto the file. EOF may not be pushed back. Returns ch or EOF on error.

`int printf(const char* format_string, ...);` Prints a string with values possibly inserted into it to standard output. Takes a variable number of arguments – first a format string followed by a number of matching arguments. The format string contains text mixed with % directives which mark things to be inserted in the output. %d = int, %Ld=long int, %s=string, %f=double, %c=char. Every % directive must have a matching argument of the correct type after the format string. Returns the number of characters written, or negative on error. If the percent directives do not match the number and type

of arguments, printf() tends to crash or otherwise do the wrong thing at run time. fprintf() is a variant which takes an additional FILE* argument which specifies the file to print to. Examples:

```
printf("hello\n");
        // prints: hello
printf("hello %d there %d\n", 13, 1+1);
        // prints: hello 13 there 2
printf("hello %c there %d %s\n", 'A', 42, "ok");
        // prints: hello A there 42 ok
```

int scanf(const char* format, ...) Opposite of printf() – reads characters from standard input trying to match elements in the format string. Each percent directive in the format string must have a matching pointer in the argument list which scanf() uses to store the values it finds. scanf() skips whitespace as it tries to read in each percent directive. Returns the number of percent directives processed successfully, or EOF on error. scanf() is famously sensitive to programmer errors. If scanf() is called with anything but the correct pointers after the format string, it tends to crash or otherwise do the wrong thing at run time. sscanf() is a variant which takes an additional initial string from which it does its reading. `fscanf()` is a variant which takes an additional initial FILE* from which it does its reading. Example:

```
{
    int num;
    char s1[1000];
    char s2[1000];
    scanf("hello %d %s %s", &num, s1, s2);
}
```

The above code looks for the word "hello" followed by a number and two words (all separated by whitespace). `scanf()` uses the pointers &num, s1, and s2 to store what it finds into the local variables.

int snprintf(char* buffer, size_t size, const char *format, ...) A version of `printf` that fills a char buffer with the resulting formatted string. The first two arguments of `snprintf` are the buffer to file and the size of the buffer. The remaining arguments are exactly like `printf`: a format string followed by any arguments to be formatted in the resulting string. There is also a `sprintf` function, but it is not "safe" since it does not include the buffer size in the set of parameters, which makes buffer overflows[1] easily possible.

int fprintf(FILE *stream, const char *format, ...) A version of `printf` that causes output to be sent to a file instead of to the default standard output. `printf` works exactly like `fprintf(stdout, ...)` since `stdout` is predefined in `stdio.h` as a `FILE *` that results in console output.

10.4 ctype.h

ctype.h includes macros for doing simple tests and operations on characters

isalpha(ch) Check whether ch is an upper or lower case letter

islower(ch), isupper(ch) Same as above, but upper/lower specific

isspace(ch) Check whether ch is a whitepace character such as tab, space, newline, etc.

isdigit(ch) Check whether ch is a digit such as '0'..'9'

toupper(ch), tolower(ch) Return the lower or upper case version of a alphabetic character, otherwise pass it through unchanged.

[1] http://en.wikipedia.org/wiki/Buffer_overflow

10.5 string.h

None of these string routines allocate memory or check that the passed in memory is the right size. The caller is responsible for making sure there is "enough" memory for the operation. The type `size_t` is an unsigned integer wide enough for the computer's address space (most likely an `unsigned long`).

`size_t strlen(const char* string);` Return the number of chars in a C string. EG strlen("abc")==3

`char* strcpy(char* dest, const char* source);` Copy the characters from the source string to the destination string.

`size_t strlcpy(char* dest, const char* source, size_t dest_size);` Like strcpy(), but knows the size of the dest. Truncates if necessary. Use this to avoid memory errors and buffer-overflow security problems. This function is not as standard as strcpy(), but most sytems have it. Do not use the old strncpy() function – it is difficult to use correctly.

`char *strlcat(char* dest, const char* source, size_t dest_size);` Append the characters from the source string to the end of destination string.

`int strcmp(const char* a, const char* b);` Compare two strings and return an int which encodes their ordering. zero:a==b, negative:a<b, positive:a>b. It is a common error to think of the result of strcmp() as being boolean true if the strings are equal which is, unfortunately, exactly backwards.

`int strncmp(const char *a, const char *b, size_t n);` Just like `strcmp`, except only the minimum of the lengths of `a` and `b`, and the value `n` characters are compared. There's also `strncasecmp` and `strcasecmp` which compare strings in a case-insensitive manner.

`char* strchr(const char* searchIn, char ch);` Search the given string for the first occurence of the given character. Returns a pointer to the character, or NULL if none is found.

`char* strstr(const char* searchIn, const char* searchFor);` Similar to strchr(), but searches for an entire string instead of a single character. The search is case sensitive.

`void* memcpy(void* dest, const void* source, size_t n);` Copy the given number of bytes from the source to the destination. The source and destination must not overlap. This may be implemented in a specialized but highly optimized way for a particular computer.

`void* memmove(void* dest, const void* source, size_t n);` Similar to memcpy() but allows the areas to overlap. This probably runs slightly slower than memcpy().

10.6 stdlib.h

`int rand();` Returns a pseudo random integer in the range 0..RAND_MAX (limits.h) which is at least 32767.

`void srand(unsigned int seed);` The sequence of random numbers returned by rand() is initially controlled by a global "seed" variable. srand() sets this seed which, by default, starts with the value 1. Pass the expression time(NULL) (time.h) to set the seed to a value based on the current time to ensure that the random sequence is different from one run to the next.

`int abs(int i);` Return the absolute value of `i`.

`void* malloc(size_t size);` Allocate a heap block of the given size in bytes. Returns a pointer to the block or NULL on failure. A cast may be required to store the void* pointer into a regular typed pointer. There is also a `realloc` function which can *change* the size of a heap-allocated block of memory. See the `man` page for details.

`void free(void* block);` Opposite of malloc(). Returns a previous malloc block to the system for reuse

`void exit(int status);` Halt and exit the program and pass a condition int back to the operating sytem. Pass 0 to signal normal program termination, non-zero otherwise.

`void* bsearch(const void* key, const void* base, size_t len, size_t elem_size, <compare_function>);` Do a binary search in an array of elements. The last argument is a function which takes pointers to the two elements to compare. Its prototype should be: `int compare(const void* a, const void* b);`, and it should return 0, -1, or 1 as strcmp() does. Returns a pointer to a found element, or NULL otherwise. Note that strcmp() itself cannot be used directly as a compare function for bsearch() on an array of char* strings because strcmp() takes char* arguments and bsearch() will need a comparator that takes pointers to the array elements – char**.

`void qsort(void* base, size_t len, size_t elem_size, <compare_function>);` Sort an array of elements. Takes a function pointer just like bsearch().

`int atoi(const char *s)` Return an integer parsed from the string s. This function is somewhat problematic since it cannot return errors if the string does not contain a parseable integer. You should generally use `strtol` (and related functions) which can return errors. See the `man` page on `strtol` for more.

`double atof(const char *)` Return a floating point number in `double` format parsed from the string s. Like `atoi` this function is somewhat problematic since it cannot return errors if the string does not contain a parseable floating point number. You should generally use `strtod` (and related functions) instead.

CHAPTER

ELEVEN

THANKS

Thanks to my COSC 301 students from Fall 2015 and 2016 for test-driving this text and for providing useful feedback. Thanks also to Aaron Gember-Jacobson, Chris Nevison, and Yasoob Khalid for pointing out various typos (and in Yasoob's case, the pull requests).

If you find any errors or typos in the book, or wish to make a suggestion for improvement, please file a bug report and/or make a pull request at https://github.com/jsommers/cbook/issues.

CHAPTER

TWELVE

COPYRIGHT

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[KR] B. Kernighan and D. Ritchie. *The C Programming Language, 2nd ed.*. Prentice-Hall, 1988. https://en.wikipedia.org/wiki/The_C_Programming_Language

[CPP] S. Prata. *C Primer Plus (5th ed.)*. S. Prata. SAMS Publishing (2005).

[C99] The home of C standards documents can be found here: http://www.open-std.org/jtc1/sc22/wg14/www/projects#9899. Specific new features in C99 are detailed (http://www.open-std.org/jtc1/sc22/wg14/www/newinc9x.htm), and the full language standard is also available (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf). Although there is an even more recent standard (C11), we largely focus on C99 (and earlier) features in this text.

[Evolution] D. Ritchie. *The Evolution of the Unix Time-sharing System*, AT&T Bell Laboratories Technical Journal 63 No. 6 Part 2, October 1984, pp. 1577-93. Available at: http://cm.bell-labs.co/who/dmr/hist.pdf

[Regehr] J. Regehr. A Guide to Undefined Behavior in C and C++, Part 1. https://blog.regehr.org/archives/213

[Lattner] C. Lattner. What Every C Programmer Should Know About Undefined Behavior #1/3. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[Horrific] http://thedailywtf.com/articles/One-Bad-Ternary-Operator-Deserves-Another

[Goto] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. Communications of the ACM, Volume 11, Issue 3, March 1968. https://dl.acm.org/citation.cfm?id=362947

INDEX

E

F

G

H

I

L

M

N

O

P